

# THE ORIC PROGRAMMER



S.M. GEE AND MIKE JAMES

## Other books of interest from Granada:

### The Apple II

#### APPLE II PROGRAMMERS HANDBOOK

R. C. Vile  
0 246 12027 4

### ATARI

#### GET MORE FROM THE ATARI

Ian Sinclair  
0 246 12149 1

#### THE ATARI BOOK OF GAMES

M. James, S. M. Gee  
and K. Ewbank  
0 246 12277 3

### The BBC Micro

#### INTRODUCING THE BBC MICRO

Ian Sinclair  
0 246 12146 7

#### THE BBC MICRO – AN EXPERT GUIDE

Mike James  
0 246 12014 2

#### 21 GAMES FOR THE BBC MICRO

M. James, S. M. Gee  
and K. Ewbank  
0 246 12103 3

#### BBC MICRO GRAPHICS AND SOUND

Steve Money  
0 246 12156 4

#### DISCOVERING BBC MICRO MACHINE CODE

A. P. Stephenson  
0 246 12160 2

#### ADVANCED PROGRAMMING FOR THE BBC MICRO AND ELECTRON

Mike James and S. M. Gee  
0 246 12158 0

#### BBC MICRO AND ELECTRON MACHINE CODE – AN EXPERT GUIDE

A. P. Stephenson and  
D. J. Stephenson  
0 246 12227 7

### The Colour Genie

#### MASTERING THE COLOUR GENIE

Ian Sinclair  
0 246 12190 4

### The Commodore 64

#### COMMODORE 64 COMPUTING

Ian Sinclair  
0 246 12030 4

#### THE COMMODORE 64 GAMES BOOK

Owen Bishop  
0 246 12258 7

#### SOFTWARE 64 Practical Programs for the Commodore 64

Owen Bishop  
0 246 12266 8

### The Dragon 32

#### THE DRAGON 32 And How to Make The Most Of It

Ian Sinclair  
0 246 12114 9

#### THE DRAGON 32 BOOK OF GAMES

M. James, S. M. Gee  
and K. Ewbank  
0 246 12102 5

#### THE DRAGON PROGRAMMER

S. M. Gee  
0 246 12133 5

#### DRAGON GRAPHICS AND SOUND

Steve Money  
0 246 12147 5

#### THE DRAGON 32 How to Use and Program

Ian Sinclair  
0 586 06103 7

### The Electron

#### THE ELECTRON PROGRAMMER

S. M. Gee and Mike James  
0 246 12340 0

### 21 GAMES FOR THE ELECTRON

Mike James, S. M. Gee  
and K. Ewbank  
0 246 12344 3

### BBC MICRO AND ELECTRON MACHINE CODE – AN EXPERT GUIDE

A. P. Stephenson and  
D. J. Stephenson  
0 246 12227 7

### Learning is Fun! 40 EDUCATIONAL GAMES FOR THE BBC MICRO AND ELECTRON

Vince Apps  
0 246 12317 6

### The IBM Personal Computer

#### THE IBM PERSONAL COMPUTER

James Aitken  
0 246 12151 3

### The Jupiter Ace

#### THE JUPITER ACE

Owen Bishop  
0 246 12197 1

### The Lynx

#### LYNX COMPUTING

Ian Sinclair  
0 246 12131 9

### The NewBrain

#### THE NEWBRAIN And How To Make The Most Of It

Francis Samish  
0 246 12232 3

### The ORIC-1

#### THE ORIC-1 And How To Get The Most From It

Ian Sinclair  
0 246 12130 0

#### THE ORIC-1 BOOK OF GAMES

M. James, S. M. Gee  
and K. Ewbank  
0 246 12155 6



# **The Oric Programmer**

Other books for Oric users

**THE ORIC BOOK OF GAMES**

Mike James, S. M. Gee and Kay Ewbank

0 246 12155 6

**THE ORIC-1**

and how to get the most from it

Ian Sinclair

0 246 12130 0



# **The Oric Programmer**

**S.M. Gee and Mike James**

**GRANADA**

London Toronto Sydney New York

Granada Technical Books  
Granada Publishing Ltd  
8 Grafton Street, London W1X 3LA

First published in Great Britain by  
Granada Publishing 1984

Copyright © 1984 by S. M. Gee and Mike James

*British Library Cataloguing in Publication Data*

Gee, S. M.

The Oric programmer.

I. Oric (Computer)—Programming

I. Title            II. James, Mike

001.64'2            QA76.8.0/

ISBN 0-246-12157-2

Typeset by V & M Graphics Ltd, Aylesbury, Bucks  
Printed and bound in Great Britain by  
Mackays of Chatham, Kent

All rights reserved. No part of this publication may  
be reproduced, stored in a retrieval system or  
transmitted, in any form, or by any means, electronic,  
mechanical, photocopying, recording or otherwise,  
without the prior permission of the publishers.



# Contents

<i>Preface</i>	ix
1 Programming and the Oric	1
What is a computer?	2
Special features of the Oric	4
Programs and programming	5
A new acquaintance	6
Using the Oric and a test card	7
Using the keyboard – a summary	11
As you go along	12
2 Principles of BASIC	13
Variables	13
Storing things in variables – LET	15
Finding out what's in a variable – PRINT	15
Arithmetic	16
Understanding expressions – the order of evaluation	17
Variables and constants – the full expression	19
A short program	19
Another way of altering variables – INPUT	20
Variables and constants as expressions	20
Describing BASIC	21
INPUT prompting	22
Mixed PRINT	23
Some sample programs	23

3	Looping and Choice – the Flow of Control	28
	The flow of control	28
	Looping – the GOTO	29
	Iteration – doing it again	30
	Getting out of a loop – the IF statement	31
	IF conditions	32
	Using IF – skip and select	34
	Using IF – conditional loops and REPEAT UNTIL	37
	The FOR statement	40
	FOR...TO...STEP	42
	Using the FOR loop	43
	IF...THEN and the colon	44
	IF...THEN...ELSE	46
	END – STOP	47
	A final example	48
	The flow of control summarised	48
4	Handling Text and Numbers	50
	Strings	50
	String expressions	51
	Another type of number – integers	56
	Arrays	57
	A word game	60
	Initialising variables – DATA and RESTORE	61
	Tape storage	63
5	Functions and Subroutines	64
	The idea of a function	64
	The Oric's functions	66
	Arithmetic functions	66
	The trigonometrical functions	68
	String functions	69
	Special functions	70
	User-defined functions – DEF FEN and FN	74



Subroutines – GOSUB and RETURN	75
Using subroutines	77
6 Low Resolution Graphics and Colour	79
Controlling PRINT	79
TAB and SPC	81
PLOT	82
PLOTting numbers	84
PLOT and the text cursor	86
Serial attributes	89
PLOTting, PRINTing and ESC	89
Default colour – INK and PAPER	92
The graphics characters – LORES	93
User-defined graphics characters	97
Double height characters	99
Flashing characters	100
Inverse colours	101
Changing a single character location	102
Using graphics in games	103
7 Sound and Games	104
PLAY, SOUND and MUSIC	104
Programming tunes	109
Resting	112
Pre-programmed sound effects	112
The sound of noise	113
PLAYing an envelope	115
Pitfalls	116
Attack the saucer – the SCRN function	117
8 High Resolution Graphics	121
The high resolution screen	121
CURSET and the graphics cursor	122
DRAW and CURMOV	124
CIRCLE	127

Dotted lines – PATTERN	128
Using colour – FILL, INK and PAPER	128
Inverse colours	132
Characters in high resolution – CHAR	132
Using and not using high resolution – GRAB and RELEASE	133
Finding out what's on the screen – POINT	133
Using high resolution graphics	134
9 Logic and Other Topics	140
Logic and the conditional expressions	140
Inside the Oric – PEEK, POKE, DEEK, DOKE, CALL and USR	143
Controlling BASIC – HIMEM, CLEAR, POP and PULL	146
10 Towards Better Programming	149
Finding bugs	149
Good programming style	151
Error trapping	153
Where next?	155
<i>Appendix: Bugs in the Oric's ROM and how to deal with         them</i>	156
<i>Index</i>	159



# Preface

Your Oric computer opens up lots of new possibilities and perhaps the most rewarding of these is learning to program it yourself. This book is intended primarily for the complete newcomer to BASIC and assumes no prior knowledge of computing. By the time you start to read it, however, you probably have already got to know how to use your Oric-1 so this book does not go into details about how to set up the machine initially. Similarly, although there is some discussion of special features of the keyboard in Chapter One, you are expected to try out the keyboard for yourself.

This book is not intended exclusively for the beginner, however, and if you are in the position of knowing some BASIC you may still find the early chapters useful. Lots of people have acquired their knowledge of BASIC in a piecemeal and haphazard way and although this usually serves them quite adequately at first, it makes it much more difficult to write well ordered, logical programs. So you may find you can gain quite a lot through picking up some theory by a quick read through Chapters Two and Three before moving on to the more machine-specific topics in the subsequent chapters. Chapter Four deals with the Oric's string and number handling, then Chapter Five looks at the way you can use functions and subroutines in your programs.

The next three chapters, about graphics and sound, are in some ways the heart of the book. Low resolution graphics are introduced in Chapter Six which also covers the Oric's colour system. Chapter Seven enables you to add sound and musical effects to your programs and Chapter Eight is devoted to high resolution graphics, a topic that is rather neglected in the Oric's manual. The final two chapters cover some advanced topics which will increase your range of programming skills.

We aim to teach you to write well structured programs but you may be surprised by our relaxed and unrestrictive approach to this.

In particular, you will find that, far from proscribing the GOTO, we actually advocate its use in writing unambiguous programs. This is neither laziness nor sloppiness on our part. Instead it is recognition of the fact that using GOTO is sometimes the *best* way of altering the flow of control.

Our book takes a very different approach from that adopted by a manual. It does more than simply tell you what commands are there for you to use. Indeed, it shows you *how* to use each command and discusses which command suits which situation. This means that as you gradually evolve your programming style you also gain a good understanding of how BASIC works.

As you will already realise, we cover a lot of ground. However, it is not meant to be hard going – it is meant to be fun. More importantly, it is not meant as a reading book – it is meant as a try-it-yourself book. There are lots of programs and program snippets included and they are all there for you to use. So stand by your Oric and enjoy learning BASIC.

Our thanks are due to Richard Miles of Granada Publishing for his encouragement and help in the preparation of this book; to Ian Sinclair and to Oric Products International for loaning us equipment.

## Chapter One

# Programming and the Oric

This book is about two topics, how to program in BASIC and how to get the best out of your Oric-1. In the early chapters the emphasis is on understanding and writing programs in BASIC. This is what is usually referred to as 'learning to program'. If you already know how to program in BASIC then you will still find that these early chapters have something to offer because they give a clear and systematic approach to BASIC, showing the logical nature of the language. Far too many people learn to program in BASIC by picking up a little knowledge here and a little knowledge there without ever managing to see it all clearly as an integrated whole. Most attempts to explain how BASIC works or should be used rely on ideas that are more appropriate in other computer languages (such as Pascal). It's as if you were learning English for the first time and the teacher insisted on showing you how similar it was to Latin! BASIC is BASIC and it is a lot easier to write good programs by going with its natural tendencies than by trying to make it into something different. As well as explaining the whys and wherefores of the various BASIC commands, it is the aim of this book to show you how to produce naturally well-structured programs as a matter of course.

After you have mastered the elements of BASIC, and even if you become an expert, there still remains the problem of how to use what you know to achieve the results that you want. Most of the later chapters in this book are concerned with just this problem and look at the Oric's sound and graphics and how to go about producing programs that are really effective.

The largest and most complex topic covered is the use of the Oric's graphics in Chapters Six and Eight. Graphics seem easy when you read a description of the BASIC commands used to manipulate the screen and indeed, apart from a few details the *commands* are simple. However, it is not the graphics commands that cause the

## 2 The Oric Programmer

trouble, it's seeing how they can be used together with the rest of BASIC to produce the effect that you want.

However, before moving on to BASIC, it is worth going over some of the ideas that are fundamental to all computers – including the Oric.

### What is a computer?

This question is one that can be answered at many levels. Whole chapters, even whole books, can be devoted to the subject. In order to use your Oric, however, you really don't have to ask this question – but if you do, the answer is interesting! If you want to know every detail of the way the Oric works then there is no choice but to learn about electronics. However, it is not difficult to gain an understanding of what a computer does and roughly how it does it without knowing anything about electronics or chips. The point is that a computer is something that would exist even if electronics had never been invented. Indeed, the first computers were built using cogs and gears and it took one hundred years before a valve (an early electronic component) found its way into such a machine. Although in practical terms the computer seems to be a product of microprocessor technology, the idea that lies behind a computer doesn't depend on the materials that you choose to build it from.

Every computer is composed of a number of parts that each perform a well-identified function. Any computer has to have some way of communicating with the outside world. In the case of the Oric this need is met by a keyboard, on which you type, and the TV screen, which the Oric can use to show you what you have typed and anything else it needs to tell you. The keyboard is an example of an *input* device and the TV screen is an *output* device.

A machine that could only receive information and pass it on unchanged wouldn't really be worth calling a computer. Rather it might be classed as a telephone or a telex! Inside every computer there has to be some mechanism that can change or *process* information before it is printed out. This mechanism usually takes the form (these days at least) of complex electronics hidden inside the computer. What we are talking about is often referred to as the *Central Processing Unit* or CPU but it also has a traditional English name that betrays the fact that computers were once made of cogs and gears – the *mill*.

In the Oric the CPU is contained in a single chip known as a 6502.



*What* exactly the 6502 does isn't of too much importance from the point of view of programming in BASIC and the *way* that it does it certainly isn't! In general, however, what the 6502 does is to perform arithmetic and other operations on information input from the keyboard and stored within the machine. What operations it does are controlled by a list of instructions called a *program*. This aspect of a computer is so important that you could almost say that a computer is a machine that will obey a list of instructions, but any sort of definition of a machine as complicated as a computer is dangerous! What sort of instructions the Oric can obey will occupy the rest of this book, so for the moment the subject will be set aside.

If a computer is going to obey a list of instructions concerning what to do with various pieces of information it must obviously have somewhere to store not only the information but also the list of instructions. This part of a computer is known as *memory* but the slightly less general term RAM (standing for *Random Access Memory*) is almost universally used instead. You can think of RAM as a sort of notepad where the CPU can record its list of instructions and any data that it needs. Obviously every memory has a limited capacity and this is an important measure of how powerful a computer is. The larger the memory, the larger the list of instructions that can be stored. The most convenient unit of measurement to apply to computer memory is the *byte*. Roughly speaking a memory that can store one byte can store one character. (Here the term character means a letter, a digit or any punctuation that you might find in normal text – such as this book!) So a 4000 byte memory could store enough characters to hold about 1½ pages of this book. The only trouble with this convenient unit of measurement is that it is a little too small. Computers normally have memories that can store thousands of characters and so it makes good sense to think in terms of thousands of characters. The unit used for this is the *kilobyte*, which is often shortened to kbyte or even just K. For various reasons however, 1 kbyte isn't 1000 bytes as its name suggests, but 1024 bytes. (You may notice that this strange number is the nearest power of two to 1000 and, as you might already know computers work in binary which is based on *two* states.) Depending on which model you've chosen, the Oric-1 has 48K or 16K of RAM. (Early computers that were used by the military to calculate missile trajectories, etc., often had less than 16K.)

This book is an introduction to BASIC. BASIC is only one of a number of computer languages, but it is the one that the Oric has

## 4 The Oric Programmer

installed in it when you buy it. It is contained in a pre-allocated part of the Oric's memory. As a computer user you cannot gain access to this part of the memory to change its contents and it is known as ROM (standing for *Read Only Memory*). Both models of Oric computer have 16K of ROM.

This combination of I/O devices, CPU and memory is all that there is to a computer. The I/O communicates with the outside world, the CPU calculates and generally processes information and the memory holds the list of instructions that the machine obeys and the data that the CPU acts on. In practice, there is one addition that we must make to this list. When you switch your Oric off it *forgets* everything stored in its RAM. To keep information stored accurately, most computer memories need a constant supply of electricity. If you switch off the supply the information is lost. This sort of memory is often known as *volatile* memory. This loss of memory is something of a problem because it implies that we have to type in the list of instructions every time that the Oric has been switched off. To overcome this difficulty, most computers have a second form of memory that is *non-volatile*. In the case of the Oric this takes the form of a standard cassette tape recorder that can be used to *save* programs and data in a form that exists even when the power has been switched off. A second advantage with this type of memory is that it is *removable*. You can record a program on a cassette and then take it out of the recorder (and even send it to someone else). The Oric is then ready for you to start on a new program or go back to an old one, which you can do by *loading* it from an earlier recorded tape.

### Special features of the Oric

Every computer system, from the largest to the smallest, is some combination of the elements mentioned above – plus extras of course – so why are there so many different versions? This is rather like asking why there are so many different motor cars – the short answer is that different designers have different ideas. Lots of the apparent differences are really to do with styling. The choice of casing and keyboard are largely governed by cost considerations and what the machine will be used for. A home computer, for example, does not need to be built to withstand extremes of temperature or humidity. Similarly, different applications need different keyboards – a numeric keypad is a must for some data

entry tasks but can be dispensed with for games playing.

The choice of CPU chip might seem to be one obvious way in which micros differ from one another. At some levels there are real differences according to which chip has been selected, but as far as the BASIC programmer is concerned the choice of CPU matters hardly at all. Other features of the hardware are much more important. In the case of the Oric the real bonuses are the colour graphics and the sound generator. The Oric's graphics are special because they use a method based on teletext graphics called *serial attributes*. This method reduces the amount of memory used to produce a colour screen display but it can be tricky to use if you are not aware of its limitations. The Oric's sound, however, presents a very wide range of sound effects and three musical tones can be used by the BASIC programmer to liven up any program.

## Programs and programming

As mentioned earlier, a computer obeys a list of instructions stored in its memory. This list of instructions is known as a program and writing such lists of instructions is known as *programming*. It is often thought that programming is an activity that started with the modern digital computers but people have been writing lists of instructions for other people to obey since writing was first invented. In this sense, programming is nothing new and can be seen in the form of recipes and knitting patterns in almost every home. Perhaps one of the best examples of *traditional* programming is written music. You can think of sheet music as being a program that will instruct a musician to play a specific tune. In fact written music is very like a computer program in that it relies on using a special language that is much more precise than ordinary language. Just one note out of place and you have a different tune! A computer program is written using a special and equally precise language. In the case of the Oric this language is BASIC, the most popular programming language in the world. Just as with written music slight changes in a BASIC program can alter its meaning completely, so it is important to realise as you learn BASIC that you must pay attention to the fine details right from the very beginning. Unlike learning English, where you can first learn words and sentences and then add punctuation, you have to take notice of every comma in a line of BASIC for it to make any sense at all!

If all this talk of strict rules is worrying you it is worth saying that

the rules are usually very simple and very regular. Unlike English there are rarely any exceptions to spelling and punctuation rules in BASIC! In addition, there are some powerful underlying ideas behind BASIC. Once you have recognised these they make it easy to understand why the rules are there at all. As you progress through this book there are therefore two types of thing that you will learn – the fine detail concerning the exact form of each BASIC statement and the general features that all programming languages share. The fine detail is important to actually getting a program working but understanding the general details is what makes the act of programming a sensible occupation.

### A new acquaintance

There are two problems with using any computer. The first is simply getting used to its idiosyncrasies. The second is writing working programs. If you have spent a little time becoming familiar with the Oric's keyboard and have loaded and saved short program examples then you might feel that you are perfectly at home with the machine and it is about time to move on to the more challenging task of programming. However, it is often the case that because a computer looks like a typewriter it is treated like one and, while this will get you somewhere, ignoring the computer's special features, such as the editing keys, will make programming hard work. For this reason it is worth giving a brief résumé of the way that the Oric is used.

Getting to know a computer is a problem that exists even if you're an expert. For although there is a lot in common between different computers, there are always enough little differences to mean that there has to be a period of *adjustment* when moving from one machine to the next. For example, nearly every computer uses a standard typewriter (or QWERTY) keyboard but most place extra but very important keys in slightly different places and this can make even the most experienced look silly at first! Now, if you're an expert, then you know that this early phase soon passes, but if you're a beginner you may panic and think that computing is always going to be this tricky! The trouble is that not being 'at home' with your computer can make easy programming ideas seem difficult.

There is no way to avoid this early barrier to programming because being on friendly terms with your computer is simply a matter of time and a matter of using it. You'll come through this rather frustrating period more easily if you bear in mind the following advice:

- (1) Separate in your mind any difficulty that you encounter in using your Oric from any difficulties that you have with programming.
- (2) Don't immediately assume that any unexpected behaviour of a program means that your Oric is illogical – computers are ruthlessly logical.
- (3) Try not to confuse typing errors with programming errors.

To help you identify this initial difficulty and overcome it, this chapter includes a short program that you should try to get running on your Oric before moving on to the rest of the book. It is presented as a complete and working program for you to use to make sure that you are over the *non-programming* problems of using the Oric. At this stage you are not expected to be able to understand how it works but you might like to return to the example after studying later chapters to see how much more of it you understand.

### Using the Oric and a test card

The program given below draws a 'test card' pattern (see Fig. 1.1) on the TV screen. Apart from being an interesting example of Oric graphics you might also find it helpful in adjusting the tuning if you are using a domestic TV set. Type in each line exactly as shown, if you make any mistakes then use the delete key to backspace and type

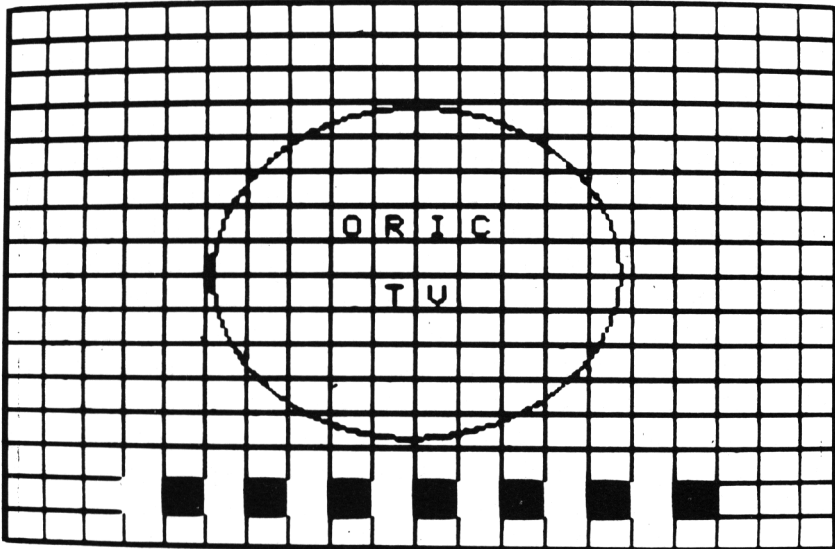


Fig. 1.1. Test card.

in your correction. When you have typed a line correctly press the RETURN key to let the Oric know that you want it added to the rest of the program. Pressing RETURN to signal that you have finished with what you are typing is a general principle. Before you press RETURN what you have typed belongs to you in the sense that you can use the DELeTe key to change it. After you press RETURN what you have typed belongs to the Oric in the sense that you can no longer change it and the Oric will examine and process it.

It is important to realise that for some purposes (capital) upper- and lower-case characters are not interchangeable. The keyboard normally produces upper-case letters but you can change this by pressing CTRL and the T key at the same time. Pressing these two keys a second time restores upper-case.

If you look at any line of the test card program you will see that it begins with a number – the *line number*. If you type something in without a line number then the Oric will attempt to interpret what you have typed as an instruction to be obeyed at once – this is called *immediate mode*. For example, once you have typed in the test card program you can examine it by typing LIST (followed by RETURN). Without a line number, LIST is taken to be a command to be acted on immediately and that is just what the Oric does – it lists the program. However, anything that starts with a line number is taken to be a command that is to be obeyed at some later time as part of a program – this is called *deferred mode*. As well as enabling the Oric to tell the difference between immediate and deferred commands, line numbers determine where a new line will be added to a program. The rule is that no matter what order you type the lines in the Oric will arrange them in order of ascending line number. So if you have already typed in lines 10 and 20, a line starting with 15 will be inserted in between them. Line numbers are also used to refer to existing lines in a program. For example, LIST 100–200 will list program lines with line numbers in the range 100 to 200. If you want to replace an existing line then simply type in your new version of the line using the existing line number. To delete a line simply type its line number followed by RETURN. To delete a block of lines you have no choice but to type in each line number in turn. If you want to delete an entire program type NEW.

Once you have typed in all of the program LIST it and check it carefully. Then type RUN and you should see the test card pattern appear. RUN is an immediate command that tells the Oric to begin obeying the lines of the program stored in its memory. To stop the program press CTRL and C at the same time. If you want to

continue the program having stopped it in this way, type CONT or simply type RUN again.

```

10 TEXT
20 INK 7
30 PAPER 0
40 HIRES
50 GOSUB 1000
60 CURSET 120,96,3
70 CIRCLE 58,1
80 GOSUB 2000
90 GOSUB 4000
100 GOTO 100

1000 FOR I=0 TO 239 STEP 12
1010 CURSET I,0,3
1020 DRAW 0,192,1
1030 NEXT I
1040 FOR I=0 TO 199 STEP 12
1050 CURSET 0,I,3
1060 DRAW 239,0,1
1070 NEXT I
1080 CURSET 239,0,3
1090 DRAW 0,192,1
1100 RETURN

2000 A$="O R I C"
2010 X=100
2020 Y=76
2030 GOSUB 3000
2040 A$="T V"
2050 X=112
2060 Y=100
2070 GOSUB 3000
2080 RETURN

3000 FOR I=1 TO LEN(A$)
3010 CURSET X,Y,3
3020 CHAR ASC(MID$(A$,I,1)),0,1
3030 X=X+6
3040 NEXT I
3050 RETURN

```



## 10 *The Oric Programmer*

```
4000 F=127
4010 Y=168
4020 FOR X=48 TO 192 STEP 24
4030 GOSUB 5000
4040 NEXT X
4050 F=1
4060 Y=168
4070 FOR X=36 TO 192 STEP 24
4080 GOSUB 5000
4090 F=F+1
4100 NEXT X
4110 RETURN

5000 CURSET X,Y,3
5010 FILL 13,2,F
5020 RETURN
```

Although you may already be familiar with how to use the Oric's keyboard it is worth making the following observations. The Oric's editing facilities are very easy to understand and can be used for purposes other than simply correcting mistakes. The text cursor, the small flashing square, is used to mark the next screen position where a character will be displayed. If you press the cursor keys (the four arrow keys to the left and right of the space-bar) you will find that you can move the text cursor without typing anything else in. If you position the text cursor over something that is already on the screen and press CTRL and A at the same time, the character under the cursor will be transferred into the Oric just as if you had typed it on the keyboard. As the text cursor automatically moves on to the next character on the line, or even on to the next line if necessary, the auto repeat facility can be used to copy long strings of characters instead of retyping them. The rule is that as long as it's on the screen you can copy it using CTRL and A. Using this idea you can add a new line to a program that is similar to one that is already included by first LISTing the existing line and then copying it, making whatever changes are necessary (including the line number). Unfortunately, this simple method appears complicated because you cannot see all you have entered until you have finished, when of course you can LIST is just like any other line of BASIC.

## **Using the keyboard – a summary**

(1) Until you press the RETURN key the Oric takes no notice of what you have typed and you can edit it using the DELeTe key.

(2) After pressing the RETURN key the Oric examines what you have typed and if it doesn't start with a line number it obeys the command at once as an immediate instruction. If it does begin with a line number then it is added to whatever program is already in memory.

(3) No matter what order the lines of a program are entered, the Oric rearranges them so that the line numbers are in strictly ascending order.

(4) The following commands are usually entered in immediate mode:

LIST start-end	Lists the program from 'start' to 'end'.
NEW	Deletes all of the program.
RUN	Causes the Oric to obey the instructions in the current program.

(5) The Oric's keyboard contains a number of special keys as well as the usual letters and digits that you would expect to see on a typewriter keyboard. In particular:

DEL	Deletes the last character typed.
Cursor keys	The four arrow keys will move the editing cursor to any position on the screen.
RETURN	Lets the Oric know that you have finished typing a line.

(6) There are a number of occasions when it is necessary to press a key at the same time as the CTRL key. In particular:

CTRL + A	Enters character under text cursor just as if it had been typed on keyboard.
CTRL + C	Stops program or listing.
CTRL + F	Key press sound on or off.
CTRL + L	Clears the screen.
CTRL + T	Switches caps lock on or off. Once the caps lock is off, pressing SHIFT together with a key produces upper-case.

(7) The ESC key is also used in conjunction with other keys but as it

## 12 *The Oric Programmer*

is used to control the colour and other features of the way that characters are displayed it is explained in Chapter Six on low resolution graphics.

### **As you go along**

As you learn BASIC and the special features of the ORIC from the rest of this book you are bound to improve, both in your use of the keyboard and your understanding of the editing features, until communicating with your Oric becomes second nature. However, until then it is wise to recall the advice given earlier in this chapter and try not to let the frustration produced by typing errors interfere with your understanding of computing in general and BASIC in particular.

## Chapter Two

# Principles of BASIC

As pointed out in Chapter One, a program is a list of instructions that your computer can carry out. The question that this poses is what sort of instructions can you use in a computer program? It is clear that instructions like ‘go and make a cup of tea’ are too vague for anything other than a human to cope with! Instructions used in a computer program must be precise. They have to specify exactly what must be done and, perhaps less obviously, they have to specify what it has to be done to. In other words a computer instruction tells the computer what to do and what to do it to. In this chapter we will look at the simplest *objects* in BASIC and some very simple things that you can do with them.

### Variables

The idea of a *variable* is the most important single idea in programming. A variable is an area of computer memory that is used to store information. This sounds like an easy idea but it has one or two subtle points. If you are going to store information in an area of memory you are going to need some way of referring to it. You’re going to have to give it a name! This is not such an unusual idea if you think about other, more traditional, ways of storing data. For example, each file in a filing cabinet is normally given a name that identifies it and it alone. Just think of the confusion of asking for a file if two files had the same name! It is just the same with BASIC; an area of memory that is used to store information, a variable, must be given a unique name that can be used to refer to it. The only additional difficulty with a BASIC variable is that you must also define what sort of objects you are going to store in the memory area. One reason for this (we will meet others later) is that the amount of memory set aside to store the information depends on

its type. For the time being, the only sort of information that we will store in memory will be numbers of any type. A variable that is used to store a number is called a *numeric variable* or a *simple variable*.

You cannot give a variable any name that takes your fancy because this would lead to confusion on the computer's part. For example, suppose you gave the name '1' to a variable. How would the Oric know the difference between the variable 1 and the number 1? In the case of the Oric you can give a simple variable a name of any length as long as it starts with an upper-case letter and thereafter uses only upper-case letters and digits. You can also insert spaces anywhere in a name to make it more readable but the Oric ignores them! In fact, it ignores everything after the first two letters of the name. There is another restriction on variable names that is more difficult to describe. If you use a variable name 'LIST' for example how is the Oric to tell the difference between 'LIST', the variable, and LIST, the command to list a program? To avoid this confusion the Oric will not allow you to use any name that includes one of its command words. The trouble is of course that you do not know all of the Oric's command words at this stage! In practice all you can do is to look out for any variable names that the Oric objects to by giving you an error message and change them to something else. Here are some examples of simple variable names that are allowed:

```
SUM
COUNT1, COUNT2
C1, C2
DAY2 MTH3 YEAR83
```

Notice that 'COUNT1' and 'COUNT2' are treated as the same names. To distinguish between two variables you need to use 'C1' and 'C2'. It is often difficult to think up names for variables that suggest the nature of the information to be stored in them but it is well worth doing. If you come back to read a program after a long time clear, obvious variable names can make it a lot easier to re-understand your own program! Even so you should try to avoid very long variable names – they can be very boring to type out over and over again in a program and there is the danger that you will inadvertently give two variables the same name by repeating the first two letters. Some examples of names that the Oric would not allow are:

<i>name</i>	<i>reason for rejection</i>
IDAY	starts with a number
DATE*	contains * which is not a letter or a digit
ANSWER?	? is not a letter or a digit
TOTAL	TO is an Oric command
SCORE	OR is an Oric command

## Storing things in variables - LET

Now that we know about variables and how to give them names it is time to discover how to store information in them. This can be done using the BASIC command LET. For example,

```
10 LET SUM=56
```

will store the number 56 in an area of memory called 'SUM'. If you recall, lines of a program are entered with *line numbers* that control their order in the list of instructions that make up the program.

This example is in fact our first program! If you enter it exactly as written nothing will happen until you type RUN when the Oric will start obeying the list of commands. In this case there is only one command and this is very easy to obey – the number 56 is stored in an area of memory called 'SUM'. If you think about it just a little more than this has happened. Before the program was run there was no area of memory called 'SUM' to store 56 in! When the Oric comes across the name of a variable that you wish to use to store something in, it checks to see if it already exists and if it doesn't it sets aside an area of memory of the right size and remembers its new name. So this innocent single line program has two effects – it creates the variable called 'SUM' and then it stores the number 56 in it.

The LET statement is such a common statement in BASIC that to save space and typing you can leave the word 'LET' out. So the line of the program could have equally well been written:

```
10 SUM=56
```

In future examples the word LET will be omitted from *assignment* statements!

## Finding out what's in a variable - PRINT

The one line example in the previous section is a little disappointing

because we have to take on trust that the Oric has actually stored the number 56 in a variable called 'SUM'. What we need is a command that will make the Oric find the variable and print its contents on the TV screen. The BASIC command with this effect is, most reasonably, called PRINT. If you add a new line, numbered line 20, to the previous example you will have the following two line program:

```
10 SUM=56
20 PRINT SUM
```

If you RUN this program you will be pleased to find that the number 56 is printed on your TV screen on the next free line.

It is important that at this point you understand exactly what is happening as a result of this two line program. Later on, when you have absorbed BASIC almost as a second language, you will understand what is going on without even thinking about it, but for now it is all too easy to read this two line program and think you understand it because it *sounds* OK! So, to recap what we have already learned, the first line creates a variable called 'SUM' and stores 56 in it. The second line finds the area of memory with the name 'SUM' and prints what is stored in it on the screen. (Notice that it is easy for a beginner to think that PRINT SUM would print the word 'SUM' on the screen. So, if you already understand why this interpretation is incorrect you are no longer a beginner!)

For the PRINT statement to work it has to be possible for the Oric to find the variable to which it refers. If for some reason you try to print a variable that hasn't been created then the Oric will obligingly create it for you and store zero in it! To see this, remove line 10 (by simply typing 10 and RETURN) and RUN the program again. Although the Oric looks after you in this way if you forget to create a variable, it is better not to fall into the habit of relying on it as other machines and other versions of BASIC are not so forgiving.

## Arithmetic

Our programs are slowly becoming more interesting but they are still a long way from being useful. We can now store numbers in variables and print out what is stored in any variable but so what! To be of any use we have to be able to change what is stored in a variable and print out something that we regard as an answer. The key to doing this lies in the idea of an *arithmetic expression*. An arithmetic



expression is nothing more than a piece of arithmetic that you haven't yet worked out. For example,  $3+6$  is an arithmetic expression that works out, or *evaluates* to 9. You can write an arithmetic expression on the right hand side of the equals sign in an assignment statement with the effect that the Oric will evaluate the expression and store the result in the variable. For example try:

```
1Ø SUM=3+6
2Ø PRINT SUM
```

You will see 9 printed on the screen. As promised the Oric has evaluated the expression and stored the result in 'SUM'.

As with most things to do with computers, there are rules governing what makes a correct expression. You can use the four operations that you should be familiar with from simple arithmetic. Addition and subtraction are indicated by the usual symbols, +, and, -, but multiplication and division use the symbols, \*, and, /. The reason for using \* to mean multiply, instead of a cross is that the traditional symbol is too easy to confuse with the letter 'X'. Some examples of correct arithmetic expressions are:

<i>expression</i>	<i>evaluates to</i>
$3+2$	5
$3*2$	6
$6/2$	3
$3+2-4$	1
$2.1+3.3$	5.4

Apart from the four usual operations of arithmetic there are two others that can be used on the Oric – the *unary minus* and the *raise to a power*. The unary minus sounds rather grand but it is simply the normal subtraction sign used in front of a single number. For example, the '-' in  $3-2$  is the normal subtraction sign, but the '-' used in  $-3$  is the unary minus. Although the same sign is used in both cases as we will see later they are treated slightly differently. The raise to a power sign is '^'. For example,  $2^2$  is read as 'two raised to the power of two', i.e. two squared, or four. The raise to a power sign is not used very often and it is mentioned here more for completeness than for its importance.

## Understanding expressions – the order of evaluation

Although the idea of an arithmetic expression seems straightforward,

there is a hidden complication. For example, if you write the innocent looking expression  $3+2*4$  does it mean 'three plus two' (i.e. five) 'times four', answer twenty, or does it mean 'three plus' the answer to 'two times four' (i.e. 'three plus eight'), answer eleven. It may seem strange to you that there are two possible ways to work out this expression because you may feel that one of the two methods is obviously correct and the other is equally obviously incorrect. However, even in arithmetic, there are no absolute answers! The correct interpretation is a matter of convention and isn't something that is handed down from high. The question of whether we do the '+' or the '\*' first in an expression like  $3+2*4$  is settled by a general agreement that multiplication is more important than addition and so it should be done first, making the correct answer eleven. This agreement that multiplication is more important than addition can be formalised in terms of assigning *priorities* to each operation and carrying out the operation with the highest priority first. The assignment of priorities can be extended to every operation that can be used in an expression (even some that we haven't met as yet). The priorities that the Oric uses to sort out the order in which arithmetic should be carried out are:

<i>operation</i>	<i>priority</i>
Unary -	1 - highest
^	2
*, /	3
+, -	4 - lowest

To evaluate an expression you should always work out the operators with the highest priority first. If two operators in an expression have the same priority then you do the one furthest to the left first (i.e. in the absence of any other preference, you work from left to right).

All this may seem a little over-complicated just to carry out a little arithmetic but it is necessary if you want to write unambiguous expressions. However there is another way of specifying the order of evaluation that can be used to override the usual priorities - brackets (). It is a longstanding convention that any parts of an expression enclosed in brackets are carried out first. For example, although  $3+2*4$  is 11,  $(3+2)*4$  is 20. If you're ever in any doubt about how the Oric will evaluate an expression then put brackets around the parts that you want worked out first. Brackets sometimes waste time and effort but they can never cause trouble!

## Variables and constants – the full expression

So far we have looked at arithmetic expressions involving only numbers but there is no reason why we cannot use variables in expressions. If you write an expression such as 'SUM+3' the Oric will find the variable called 'SUM' and retrieve the number stored in it. It will then add three to this number. For example, if 'SUM' had 32 stored in it, the expression 'SUM+3' would evaluate to 35. Notice that there is no suggestion that what is stored in the variable 'SUM' is in any way altered. Its contents are simply used in the evaluation of an expression. A number such as 3 is known as a *constant* (because its value never changes!) and now we can see that an expression can be made up of variables and constants with the arithmetic operators, +, -, /, \*, ^. An expression always evaluates to a constant and it is this constant that is stored in a variable by an assignment statement.

## A short program

Using all that we have found out so far about constants, variables and expressions we can now write a short program that adds two numbers together:

```
1Ø N1=23.34
2Ø N2=44.32
3Ø ANSWER=N1+N2
4Ø PRINT ANSWER
```

If you enter and RUN this program you will see that the sum of the two numbers in lines 1Ø and 2Ø are printed by line 4Ø. Although this is another simple example it demonstrates a wide variety of programming ideas. Lines 1Ø and 2Ø store two constants in two variables. In line 3Ø the arithmetic expression 'N1+N2' is evaluated and the result is stored in a third variable 'ANSWER'. Line 4Ø prints the contents of 'ANSWER' on the screen. If you think this is easy so far so good! Try changing lines 1Ø and 2Ø to add different numbers together and change line 3Ø to give you different arithmetic expressions.

**Another way of altering variables - INPUT**

In the previous example the two variables 'N1' and 'N2' had numbers stored in them by use of the assignment statement. This is convenient unless we want to use the program many times with different values. As suggested, the only way that it is possible to change the values stored in the variables is to edit each line before running the program. Obviously what we need is a statement that will allow us to enter any value into the variable *while the program is running*. This is what the BASIC statement INPUT is for. For example try the following program:

```
1Ø N1=5
2Ø INPUT N2
3Ø ANSWER=N1+N2
4Ø PRINT ANSWER
```

When you run this you might be surprised to find that nothing much happens! *Don't panic!* What has happened is that line 1Ø was carried out and 5 was stored in the variable 'N1'. Then the Oric moved on to line 2Ø where it obeyed the command INPUT by waiting for you to type a number and this is why the question mark was displayed. The Oric is waiting for you to type a number and then press RETURN to signal that you have finished typing/correcting the number. It then stores the number that you have typed in the variable 'N2' and proceeds to the next instruction. So if you haven't already done so, run the program and type in a number of your choice. You will be pleased to see your number with five added to it printed in the usual place.

We now have two ways of storing numbers in a variable – simple assignment and INPUT. It is important to understand the difference between the way these two methods work. As in the case of simple assignment, if a variable doesn't exist before its use in an INPUT statement the Oric will create it. If you always want to store the same value or the result of an expression in a variable then use an assignment statement. If you want to store a different value in a variable each time the program is run then use an INPUT statement.

**Variables and constants as expressions**

One of the most powerful features of BASIC is the way that almost anywhere that you can use a constant or a variable you can use an

expression as well. For example, in the PRINT statement you can write:

```
30 PRINT N1+N2
```

and the Oric will evaluate the expression and print the result.

It is also true that the simplest forms of an expression are the constant and the variable. For example, the number 3 can be thought of as either a constant or an extremely simple expression. Similarly, the variable 'SUM' can also be thought of as an expression. So not only can you use an expression wherever you might use a variable or a constant, you can use a variable or a constant anywhere that you can use an expression! For example:

```
LET N1=N2
```

and

```
PRINT 3
```

are both valid BASIC statements.

## **Describing BASIC**

It is difficult to describe any language and BASIC is no different. The trouble is that while it is easy to give an example of what is correct it is difficult to explain all the possible correct variations. For example, at the start of this chapter the LET statement was introduced by:

```
LET SUM=56
```

but this gave no hint that you could write things like:

```
LET SUM=NUMBER
```

or

```
LET SUM=N1+N2
```

To try to overcome this difficulty it is usual to give a definition involving the general types of things that a statement allows. For example, the general form of the LET statement can be written as:

LET 'simple variable' = 'arithmetic expression'

where the things between the single quotes are not to be taken literally, but replaced by an example of the stated type. So in a real

LET statement 'simple variable' would be replaced by a variable name such as COUNT, SUM, NUMBER, etc.

Throughout the rest of this book BASIC statements will be introduced by examples and then defined in the same way as the LET statement above. As we learn more about a statement it may prove necessary to redefine it to include a wider range of features. So far the two other BASIC statements that we have introduced, PRINT and INPUT, can be defined as follows:

PRINT 'arithmetic expression'

and

INPUT 'simple variable'

but as we shall see later these are not the final definitions!

### INPUT prompting

Although we can now use INPUT to store information in variables, the way the Oric just stops and waits for someone to type in a number is a little unsatisfactory. What is required is the ability to print a message on the screen saying something like "TYPE IN A NUMBER NUMBER NOW" or "WHAT IS YOUR NUMBER". Such a message is often called an *input prompt* and BASIC provides two similar ways to print such messages. Try the following short program:

```
1Ø PRINT "THIS IS A PROMPT"
2Ø INPUT "WHAT IS YOUR NUMBER ?";N1
```

Line 1Ø will display "THIS IS A PROMPT" on the screen and line 2Ø displays "WHAT IS YOUR NUMBER ?" just below it and then waits for you to type a number. In both cases the characters printed on the screen are the ones inside the double quotation marks. A set of characters in double quotes is known as a *literal string* or simply as a *string*. You can use either PRINT or INPUT to produce prompt messages on the screen depending on which is more convenient. As an example of the use of both, consider the following version of the number addition program given earlier:

```
1Ø PRINT "WHAT IS YOUR FIRST NUMBER ?"
2Ø INPUT N1
3Ø INPUT "WHAT IS YOUR SECOND NUMBER ?";N2
4Ø PRINT N1+N2
```

## Mixed PRINT

The ability to print messages on the screen is clearly a very useful facility for other things than just printing prompts. For example, in the last program it would have been better to print a message saying that the number about to be printed was the sum of the two numbers. You can in fact use a single PRINT statement to print more than one thing at a time. For example, change line 40 in the previous program to:

```
40 PRINT N1;" + ";N2;" = ";N1+N2
```

and you will see that the contents of 'N1' are printed then a space and a plus sign followed by another space then the contents of 'N2' followed by an equals sign with a space on either side of it and the answer. You can consider this PRINT statement as a list of items to be printed with each item in the list being separated by a semicolon and printed in the next free printing position. Our general definition of PRINT can now be updated to read:

PRINT 'print list'

where 'print list' is a list of items separated by semicolons. The items can be either expressions or strings. Each PRINT statement starts printing on a new line. In Chapter Six we will return to the definition of the 'print list' and expand it to include ways of formatting the information on the screen but this simple version of the 'print list' will satisfy all our requirements until then.

## Some sample programs

Even with so little BASIC it is possible to write some useful, if simple, programs. For example, if you want regularly to work out how many dollars you could buy for a given number of pounds then a currency conversion program would be useful. The overall outline of this program is easy to explain as follows:

```
ask the user for the conversion rate
ask for the number of pounds to be used to buy dollars
print number of pounds times conversion rate
```

By now you should realise that this program is a simple one for your Oric, so before you look at the version given below, try to write your own version. Remember to include input prompting and explanatory

messages. There is no one perfect way to write any program, so don't worry if your version is different from the one given. It could well turn out to be better!

```

10 PRINT "Pounds to Dollar conversion"
20 INPUT "What is the conversion rate";RATE
30 INPUT "How many pounds do you want to
    spend ";AMOUNT
40 PRINT
50 PRINT "For ";AMOUNT;" pounds "
60 PRINT "you can buy ";RATE*AMOUNT;" dollars"

```

Line 10 simply prints a title for the program. Lines 20 and 30 prompt for and accept the necessary input. Notice how line 40 is used to print a blank line to space the output into an input and a result section. Lines 50 and 60 print the answer with some explanation.

As another example, consider the problem of calculating the stopping distance of a car travelling at any speed in miles per hour. The main problem in writing this program is knowing how to calculate the stopping distance. It is important to realise at this stage that no computer can calculate something unless you can explain to it *how* to do the calculation. Looking at the highway code reveals that the stopping distance is made up from two components – a thinking distance and a braking distance. The thinking distance in feet is roughly the same as the speed in mph. The braking distance is a little more complicated and is given by the square of the speed divided by 20. However both of these quantities are very easy for the Oric to calculate, so to add to the usefulness of this short program it is reasonable to print out the thinking distance, the braking distance, the overall distance and how many car lengths it takes to come to a stop. This last calculation is based on the fact that the average (UK) car is 14 feet long. The program is now easy to write:

```

10 PRINT "Stopping Distance"
20 PRINT
30 INPUT "Speed in mph=";SPEED
40 PRINT "at ";SPEED;"mph"
50 PRINT
60 PRINT "Thinking distance=";SPEED;" feet"
70 BRAKE DIST=SPEED*SPEED/20
80 PRINT "Braking distance=";BRAKE DIST;" feet"

```



```

90 PRINT "Overall distance=";SPEED+BRAKE
  DIST;" feet"
100 PRINT
110 PRINT "Which is ";(SPEED+BRAKE DIST)/14;" car
  lengths"

```

Line 10 simply prints a title for the program and line 20 provides a line of space beneath it. Line 30 prompts for the only information needed by this program – the speed in mph. This is stored in the variable 'SPEED'. Lines 40 and 50 start giving the answer to the user by printing the speed that the calculation is for and leaving some space. Line 60 prints the thinking distance, which requires no calculation as it is numerically the same as the speed in mph. The braking distance is calculated by line 70 and printed by line 80. The overall distance is calculated and printed by line 90. Notice that this is an example of using an expression in a PRINT statement. Line 100 leaves another space before line 110 calculates and prints the overall stopping distance in terms of car lengths. Notice that you have to put brackets around the addition to make sure that it is carried out first.

The final example in this chapter is a sizable and really useful program if you are interested in DIY. A very common problem is that of estimating how much sand, aggregate and cement you should buy to cast a slab of concrete. To help with this difficult task the following program is a "Concrete Calculator". Once again our first problem is knowing how to do the calculation before we begin writing the program. Looking up the relevant information in a book on building reveals the following – 1 cubic metre of concrete made up from 1 part cement,  $x$  parts of sand and  $y$  parts of aggregate (i.e. a 1: $x$ : $y$  mix) needs

$$\frac{1}{0.025*(1+x+y)}$$

bags of cement and

$$\frac{x*1.5}{(1+x+y)}$$

cubic metres of sand and

$$\frac{y*1.5}{(1+x+y)}$$

cubic metres of aggregate. (It's not too important to understand why

these equations work, it is often the case that a program is written from information that the programmer doesn't fully understand – and why should it be otherwise!) Converting this information into a program is once again relatively easy. The first part of the program should ask for the dimensions of the concrete slab and then calculate its volume. The program should then ask for the mixture ratio and using the equations given above, work out the number of bags of cement and the volume of sand and aggregate required. Finally, the results should be printed out in a form that the user will find acceptable. The details of the program are:

```

10 PRINT "Concrete Calculator"
20 PRINT
30 INPUT "What is the thickness in mm ?";MMT
40 INPUT "What is the length in m ?";ML
50 INPUT "What is the width in m ?";MW
60 PRINT
70 VOL=MMT*.001*ML*MW
80 PRINT "Total volume = ";VOL; " cubic m"
90 PRINT
100 INPUT "How many parts of sand to one of
    cement ";SPART
110 INPUT "How many parts of aggregate to one of
    cement ";AGGPART
120 SUM=1+SPART+AGGPART
130 CEMENT=VOL/SUM/.025
140 TS=VOL*SPART*1.5/SUM
150 TAGG=VOL*AGGPART*1.5/SUM
160 PRINT "Using a 1:";SPART;" "AGGPART;" mix"
170 PRINT "you need "
180 PRINT CEMENT;" bags of cement"
190 PRINT TS;" cubic m of sand"
200 PRINT "and ";TAGG; " cubic m of aggregate"

```

Lines 30 to 80 ask for the dimensions of the slab and both calculate and print the total volume. Lines 100 and 110 ask for the ratio of the mix and line 120 calculates  $1+x+y$  used in all of the following calculations. The most interesting lines of the whole program are 130 to 200. The first few lines (130–150) carry out the main calculations and the last section (160–200) print the results. If you look carefully at the calculations you will see that an arithmetic expression in BASIC doesn't always look like the equation that it comes from. For example, you might fall into the trap of writing,

$$\frac{1}{0.025*(1+x+y)} \text{ as } 1/0.025*(1+x+y)$$

but the result of this BASIC arithmetic expression is given by dividing 1 by 0.025 and then multiplying the answer by (1+x+y) i.e. it works out:

$$\frac{1}{0.025} *(1+x+y)$$

rather than the equation that we are interested in. The correct expression is:

$$1/0.025/(1+x+y)$$

or if you want to use extra brackets to clarify matters:

$$1/(0.025*(1+x+y))$$

In the section that prints the results notice the way that each answer is *embedded* in the printed message. Remember that you do not have to write programs that produce results in a standard way – experiment with what looks good and seems natural.

Before moving on to the next chapter you might like to find a calculation like the one carried out by the Concrete Calculator that is often used either in your home or at work, and write a program to automate it. Two suggestions are the cost of a telephone call, given the time and unit charge, and the amount of electricity consumed by a device, given its wattage and the time it is used for (1000 watts for 1 hour = 1 unit of electricity).

## Chapter Three

# Looping and Choice - the Flow of Control

So far we have written programs that are lists of instructions which are carried out one at a time from the top to the bottom. Although it is possible to write useful programs using nothing more, programming really only becomes interesting when you can change the order in which instructions are carried out.

### The flow of control

If you look at any of the example programs at the end of Chapter Two it should be possible for you to follow through with your finger the order that the instructions would be obeyed by the Oric. You can think of this as tracing the *flow of control* through the program. Each instruction has its turn at governing or *controlling* what the Oric is doing and it then *passes control* to the next instruction. In the absence of any other information the next instruction is taken to be the next line down the listing or, put another way, the line with the next highest line number. So the *default* flow of control is a line starting at the top of the program and finishing at the bottom. The following is a very simple program that demonstrates this default condition.

```
10 INPUT A
20 PRINT A
30 INPUT B
40 PRINT B
```



Fig. 3.1.

## Looping – the GOTO

BASIC provides a single statement for changing the default flow of control – the GOTO statement. Try the following program:

```
10 TEST=0
20 PRINT TEST
30 GOTO 20
```

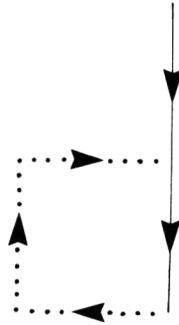


Fig. 3.2.

If you run this program you will see the screen fill with zeros (one to each line). After this you may not be able to see the zeros being printed but they still are! The only way that you can stop the program, without losing it, is to press CTRL and C simultaneously.

This program is our first example of a *loop*. Tracing the flow of control through the program soon shows why the word *loop* is appropriate. First line 10 stores zero in the variable 'TEST', then line 20 prints the contents of 'TEST'. Line 30 is new in that it uses the command GOTO but its meaning should be clear from just reading it. The statement GOTO 20 causes the Oric to obey line 20 as the next instruction. So after line 30 control is transferred to line 20 and the contents of variable 'TEST' are printed on the screen for a second time. After this line 30 is again carried out. This transfers control back to line 20 and so on ... The program calls for the repetition of lines 20 and 30 forever and tracing the flow of control shows it to take the form of a loop. It is not a very useful loop, however, and because it has no predetermined stopping point it is usually called an *infinite loop*.

A GOTO statement can be used to force the Oric to carry out any instruction next. Its general form is:

GOTO 'line number'

For example:

GOTO 20

You may be wondering what happens if you write something like

GOTO 50 and line 50 doesn't exist. In this case the Oric will stop and give an error message "?UNDEF'D STATEMENT ERROR IN X" (where X is the number of the line that contains the GOTO) indicating that you are trying to transfer control to a non-existent line.

### Iteration - doing it again

Although the example of the infinite loop serves to introduce the idea of transferring control to a different point in the program, it doesn't really indicate the sort of thing that a loop is used for. An important idea in programming is the repeating of series of operations. This is often called *iteration*. For example, the instruction NUM=NUM+1 simply adds one to the contents of the variable 'NUM' and stores the answer back in 'NUM'. In other words it increases the number stored in 'NUM' by one. If you repeat this operation by using a GOTO you have something more than adding one to a variable - you have a program that counts! Try the following:

```
10 NUM=0
20 NUM=NUM+1
30 PRINT NUM
40 GOTO 20
```

you will see the screen fill with numbers without end. If you look carefully you should be able to see that the numbers are increasing by one each time - your program is counting. Notice that by adding a GOTO to an instruction that adds one to a variable we actually seem to produce a program that does a bit more than just count. In fact, we generate a *sequence* of numbers. This is much more the flavour of real programming than the one-after-the-other programs in Chapter Two. To see looping doing something a little more useful try:

```
10 NUM=0
20 PRINT "x =";NUM;" x squared =";NUM*NUM
30 NUM=NUM+1
40 GOTO 20
```

This program will print out two lists of numbers, the second being the square of the first.

Even if you are very familiar with the idea of a loop you can be

confused about what the current value of a variable is at any point in the loop. For example, in the case of the counting loop program, the first value of 'NUM' that was printed was one but in the squares program the first value was zero. This difference is simply due to *where* in the program the line that adds one to 'NUM' is placed and *also* what value 'NUM' is set to before the loop starts. If you change line 10 in the squares program to read 10 NUM=1 then the first value to be printed will be one.

Understanding what goes on in a loop gets easier with practice but it's all a matter of following through the action of the program clearly and without rushing. Whenever you try to write or understand a loop, identify the start of the loop and its end, then you will be absolutely clear about which instructions are repeated.

### Getting out of a loop – the IF statement

Although the GOTO is a very useful statement the only thing that you can do with it is to form infinite loops. This is fine for simple things like printing tables of values but is a bit too crude for many applications. What we are lacking is a statement that will stop the loop when a *condition* is satisfied. For example, suppose we want to write a program that will add ten numbers together and then print out the answer. At the moment the best that we can do is to read in each number in turn and add it to a *running total* which we print out each time, as in the following example:

```

10 SUM=0
20 NUM=0
30 INPUT "Number = ";NO
40 SUM=SUM+NO
50 NUM=NUM+1
60 PRINT NUM;" Total = ";SUM
70 GOTO 30

```

If you run this program you will find that it produces rather a lot of output that you don't need. What we really want to do is read in the ten numbers, keep a *running* sum and only print out the answer at the end. This can be achieved using the IF statement:

```

10 SUM=0
20 NUM=0
30 INPUT "Number = ";NO

```

```

40 SUM=SUM+NO
50 NUM=NUM+1
60 IF NUM=10 THEN GOTO 80
70 GOTO 30
80 PRINT "Total = ";SUM

```

The only difference between this and the first program to add ten numbers together is the use of the IF statement in line 60. Each time through the loop formed by the lines from 30 to 70 the IF statement is obeyed. This takes the form of comparing the contents of the variable 'NUM' to 10. If it isn't equal to 10, control passes to the next statement, i.e. line 70, and the GOTO following the THEN has no effect. However, if 'NUM' is equal to 10 the GOTO following the THEN is carried out and control passes to line 80 printing the answer and ending the loop. You should be able to see that there are now two paths through this program; the first is the loop which is taken if NUM is anything but 10 and the second is the 'jump out of the loop' produced by the IF statement only when NUM is 10. The task of tracing the flow of control through this program is clearly made more difficult by there being more than one way through it but you can still trace each of the possible routes.

## IF conditions

There are many ways of using the IF...THEN GOTO statement to alter the flow of control depending on whether or not a *condition* is true. Before we can go on to investigate the sort of thing that can be done with IF we have to find out what types of conditions we can use.

All of the conditions that you can use in an IF statement take a very simple form:

'arithmetic expression1' 'relation' 'arithmetic expression2'

We already know what an *arithmetic expression* is so the only new element is the *relation*. In BASIC there are six relations:

<i>relation</i>	<i>meaning</i>
=	equals
>	greater than
<	less than
<=	less than or equal to
>=	greater than or equal to
<>	not equal to



The meaning in BASIC of each of these relations is the same as their normal meaning.

Conditions are very often called *conditional expressions* and a condition is like an expression in that it evaluates to a value but in its case there are only two possible results, true or false. It is important to read conditions in the right way to avoid confusion. For example, the condition 'NUM=3' is *not* an instruction to make NUM equal to 3, it is a *question* about what is stored in 'NUM'. If the value stored in 'NUM' is 3 then 'NUM=3' is true but if the value is anything other than 3 then 'NUM=3' is false. Thus the Oric uses the sign = in two different ways – as an instruction to store a value in a variable and as a relation in a condition. The only way to tell which is the correct meaning in any case is to look at the rest of the instruction. Some examples of conditions are:

NUM<>4	false if NUM is 4, otherwise true
NUM*2>10	true if NUM*2 is greater than 10, otherwise false
3>1	ALWAYS true
1>6	ALWAYS false

To reinforce the idea that a condition is an expression that evaluates to one of two values (true or false) it is worth saying that the Oric represents both values as numbers. True is represented by -1 and false is represented as 0. In other words, if a condition is true it evaluates to -1 and if it is false it evaluates to 0. To prove that this is the case try the following program:

```

10 INPUT "First number";A
20 INPUT "Second number";B
30 PRINT "The result of ";A;">";B;" is ";A>B
40 GOTO 10

```

You will find that either a -1 or a 0 will be printed depending on whether 'A>B' is true or false in the case of the numbers you typed. You might be surprised that you can write a condition in a PRINT statement where you would normally write an arithmetic expression. This is simply another reflection of the fact that a condition is an expression just like an arithmetic expression and it can be used anywhere that an arithmetic expression can. Indeed, you can mix conditional and arithmetic expressions with no problems as long as you pay attention to the order in which they are evaluated. Any arithmetic is evaluated before any conditional expressions. For example:

```
PRINT (2<1)+(3>1)+(3=2+1)
```

will print -2 on the screen because (2<1) is false and evaluates to 0, (3>1) is true and evaluates to -1 and (3=2+1) is also true and evaluates to -1, which gives 0-1-1 i.e. -2. (Notice that the conditions all have to be enclosed in brackets to stop the Oric trying to do the arithmetic first!) This sort of thing is fairly advanced BASIC so don't worry if you don't understand it fully - what is important is that you understand that something like 2=3 is an expression and evaluates to true or false rather than an instruction to do some operation.

We have taken a slight detour around the subject of the IF statement to examine the idea of a conditional expression but armed with this new information the remainder of this chapter should seem easier. The general form of the IF...THEN GOTO statement is:

```
IF 'conditional expression' THEN GOTO 'line number'
```

If the conditional expression evaluates to -1, or true, then the GOTO following the THEN is obeyed. If the conditional expression evaluates to 0, or false, then the statement following the IF is obeyed.

There are so many ways of using the IF...THEN GOTO apart from breaking out of loops that it is difficult to give examples of everything. But if you understand the ideas of flow of control and the way that IF and GOTO can be used to change it then you should have no trouble in understanding the examples in the rest of this book.

#### **Using IF - skip and select**

The only way to find out how useful IF can be is to write your own programs that use it. In this way you'll slowly pick up all the standard ways that you can change the flow of control depending on the result of a conditional expression. However, giving examples of some of the most common ways that it is used may help to speed up this process and avoid clumsy use of the IF statement. The idea of using an IF statement to break out of a loop has already been introduced and this is so important that it deserves a number of sections all to itself! However, apart from its use in loops, the IF statement can be used to *skip* a section of program and *select* between a number of sections. For example:

```

10 INPUT A
20 IF A>0 THEN GOTO 40
30 A=-A
40 PRINT A

```

This short program refuses to let you enter a negative number! The IF statement in line 20 checks to see if the number in 'A' is greater than zero and if so control passes to line 40 – effectively skipping line 30. If 'A>0' is false line 30 changes the sign of the contents of 'A'. In another program a different list of statements might be skipped according to some other condition. The *shape* of the skip is depicted in Fig. 3.3 which illustrates how the result of the condition decides whether or not the list of instructions is carried out or skipped.

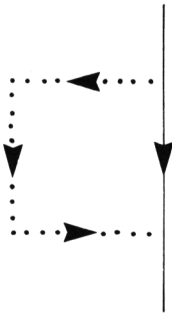


Fig. 3.3.

An extension of the idea of skipping some lines of BASIC is to choose between two different lists of commands. In this case it is easier to understand the idea after looking at the shape of the flow of control (see Fig. 3.4). Which list is carried out depends on the condition used in the IF statement. It is useful to look at Fig. 3.4

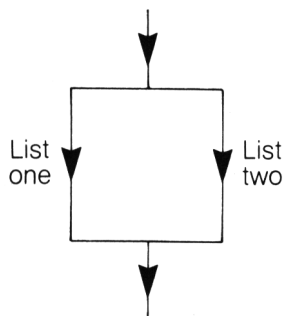


Fig. 3.4.

before looking at an example of an IF statement to select between two sets of instructions because it may be little difficult to see the simple division into two in the BASIC. Consider the following program:

```

10 INPUT A
20 IF A<0 THEN GOTO 60
30 PRINT "A is positive"
40 B=A
50 GOTO 80
60 PRINT "A is negative"
70 B=-A
80 PRINT A,B

```

According to the value of 'A' either lines 30, 40 and 50 are obeyed or lines 60 and 70 are. The *division* point in the diagram corresponds to the IF statement itself (line 20). The *join up* point is line 80 because this is the first statement that will be carried out no matter which of the two lists of instructions is selected. If you try to superimpose Fig. 3.4 on the program you will see that, although it represents what happens, it is difficult to fit it. The reason for this is that it is impossible in BASIC to write the two alternative lists next to each other so the flow of control diagram is more like the one in Fig. 3.5.

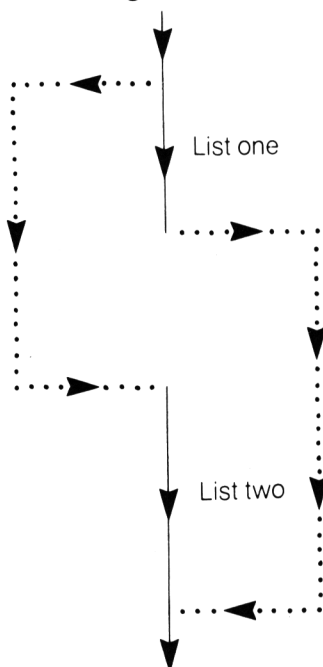


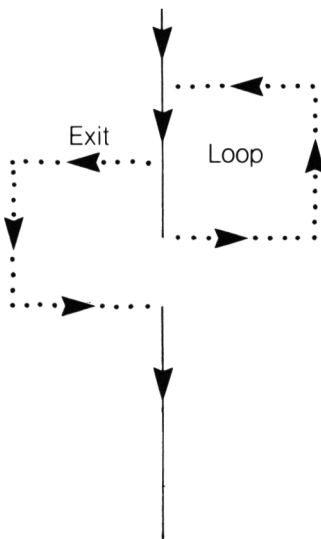
Fig. 3.5.

If you look at Fig. 3.5 you should be able to see that it is a ‘mangled’ form of Fig. 3.4. Using the IF to select between two alternatives is easier to understand from Fig. 3.4 but Fig. 3.5 corresponds more closely to reality. This is the most complicated flow of control diagram that you will find in BASIC and to make it slightly simpler Oric BASIC includes a special form of the IF statement – IF... THEN...ELSE – which will choose between two alternatives. This is a simple statement but we need a few more ideas before it becomes useful to us so a discussion is left to the end of this chapter.

There are many other ways of using the IF and GOTO statements to alter the flow of control in a program and there is no need for you to feel restricted to just the skip and the select. However, this is one area where it is not wise to be too adventurous because a program that changes the flow of control in a haphazard way will be very difficult to read and therefore likely to contain bugs that are hidden in the knots that you have tied in the flow of control! The rule is that in a well written program it should be easy to trace the flow of control and restricting yourself to only a small number of ways of transferring control is one way of keeping things simple.

### Using IF – conditional loops and REPEAT UNTIL

We have already met the other important use of the IF statement,



*Fig. 3.6.*

that of breaking out of a loop. The flow of control diagram for that circumstance can be visualised as shown in Fig. 3.6.

You should be able to see the familiar shape of an infinite loop. The path that leads out of the loop and back to the normal flow of control corresponds to a GOTO taken when the condition in the IF statement is true. A loop that ends when a particular condition is true is known as a *conditional loop*. Notice that the point at which the IF statement breaks out of the loop can be placed anywhere. In other words, you can break out of a loop anywhere from the first statement to the last. For example:

```
10 A=0
20 A=A+1
30 IF A=20 THEN GOTO 60
40 PRINT A
50 GOTO 20
60 PRINT "Finished"
```

has its exit point in the middle and

```
10 A=0
20 A=A+1
30 PRINT A
40 IF A=20 THEN GOTO 60
50 GOTO 20
60 PRINT "Finished"
```

has its exit point at the end of the loop.

In general, although it is possible to place the exit point of a loop anywhere, it is better to place it either right at the beginning or right at the end. The reason for this is that it is better to avoid carrying out *part* of a loop. Loops with an exit point either at the beginning or at the end always repeat all of their instructions the same number of times and this makes understanding programs very much easier. A loop with an exit point at the beginning may exit without ever obeying the instructions within the loop but a loop with an exit point at the end must execute the instructions within the loop at least once. This observation can be used to decide where you need to place the exit point in a conditional loop to produce the result that you desire. If you know what there are occasions where the loop should be skipped completely then place the exit point at the start, but if you always want to carry out the loop at least once then place the exit point at the end.

Technically a loop with only one exit point placed at the

beginning is known as a *while* loop and a loop with only one exit point at the end is known as an *until* loop. Some programming languages include extra statements to encourage the use of while or until loops as opposed to sloppy conditional loops with exit points placed anywhere. Most versions of BASIC are happy to leave the programmer to construct whatever conditional loops he or she needs, using the IF...THEN GOTO statement to break out, and hope that good programming style alone will encourage programmers to be careful about where such statements are placed. However, Oric BASIC does provide two extra statements to make loops with exit points at the end particularly easy to use – REPEAT and UNTIL. The statements REPEAT and UNTIL have to be used as a pair in that REPEAT is used to mark the start of a conditional loop and UNTIL marks its end. The statements that lie in between a REPEAT...UNTIL pair are obeyed repeatedly just as in a standard loop. The full form of UNTIL is:

UNTIL 'condition'

where 'condition' is a conditional expression as used in an IF statement and the loop will end when the 'condition' is true. The action of the REPEAT...UNTIL loop is accurately reflected in the English reading of:

```
1Ø REPEAT
2Ø . . .
3Ø list of other BASIC statements
4Ø . . .
5Ø UNTIL condition
```

in that the 'list of other BASIC statements' is indeed repeated until the 'condition' is true. Notice that the REPEAT...UNTIL pair does not introduce anything new to BASIC. It simply provides another way of writing a conditional loop with an exit point at the end. For example:

```
1Ø A=Ø
2Ø A=A+1
3Ø PRINT A
4Ø IF A=2Ø THEN GOTO 6Ø
5Ø GOTO 2Ø
6Ø PRINT "Finished"
```

can be written as

```
10 A=0
20 REPEAT
30 A=A+1
40 PRINT A
50 UNTIL A=20
60 PRINT "Finished"
```

Using REPEAT to mark the start of the loop and UNTIL to break out. Although in this sense REPEAT ... UNTIL is redundant, it is nearly always to be preferred to using IF ... THEN GOTO to construct *until* loops. The reason for this is that if you look at a program and spot an IF statement you still have to work out what it is being used for – skip, select or conditional loop. However, if you see a REPEAT statement you know at once that you are dealing with a conditional loop and once you have located the matching UNTIL you also know the condition that makes the loop end. It should be obvious that a REPEAT...UNTIL loop should not contain an IF...THEN GOTO instruction that transfers control out of the loop!

### The FOR statement

Apart from the position of the exit point, loops differ in two other ways. All loops continue until a condition is satisfied but in many cases this is equivalent to carrying out the loop a fixed number of times. For example, if you wanted to print the word “Hello” on the screen five times you could do it using:

```
10 A=1
20 REPEAT
30 PRINT "Hello"
40 A=A+1
50 UNTIL A>5
```

Loops that cause a list of instructions to be repeated a given number of times are called *enumeration loops* – (enumeration means to count). However, this is such a common situation that BASIC provides two extra statements – FOR and NEXT – to make repeating lists of statements easier. Using FOR and NEXT the program that prints “Hello” on the screen five times can be written:



```
10 FOR A=1 TO 5
20 PRINT "Hello"
30 NEXT A
```

The meaning of the FOR and NEXT should be clear from the program. The variable 'A' is used to count the number of times that the loop has been obeyed in the same way as in the earlier example. The difference is that everything is done automatically. The FOR statement first sets 'A' to one. Each time the NEXT statement is carried out one is added to 'A' and, as long as its value hasn't exceeded 5, control is transferred back to line 20 (i.e. there is an implied GOTO 20). The result is that the PRINT statement at line 20 is executed five times before control passes on to the statement following the NEXT.

The general form of the FOR...NEXT loop is:

```
FOR 'index variable'='start value' TO 'end value'
.      .      .      .
.      .      .      .
NEXT 'index variable'
```

The 'index variable' is initially set to the 'start value'. Each time NEXT is reached the 'index variable' is increased by one and as long as the value hasn't exceeded the 'end value' control is transferred to the statement just after the FOR. To make sure that you understand exactly what a FOR loop can do try the following examples:

```
10 FOR I=1 TO 10
20 PRINT I
30 NEXT I

10 FOR Z=100 TO 110
20 PRINT Z
30 NEXT Z
```

In general both the 'start value' and the 'end value' can be full of arithmetic expressions but it is important to realise that these are only evaluated *once* at the start of the loop. This becomes clear if you think of the FOR statement as only being carried out once at the start of the loop. If 'start value' is bigger than 'end value' when the FOR statement is first encountered the Oric will still carry out the statements between the FOR and the NEXT *once*. For example:

```
10 FOR I=10 to 5
20 PRINT I
30 NEXT I
```

will print 10 on the screen. In other words the FOR loop is another example of a loop with its exit point at the end!

The value of the 'index variable' can be used in arithmetic expressions during the loop but *its value must not be changed*. In other words, you can use the 'index variable' in a FOR statement on the right hand side of an = sign but not on the left. For example, the following short program will print a multiplication table:

```
10 INPUT "Which table - enter 2 for two times table etc";T
20 INPUT "Starting at ";S
30 INPUT "Ending at ";E
40 FOR I=S TO E
50 PRINT I;" x ";T;" = ";I*T
60 NEXT I
```

Notice that both the start and end values of the FOR loop are arithmetic expressions, simple variables in fact! Also note the use of the 'index variable' 'I' in line 50.

## FOR...TO...STEP

The simple FOR loop serves for most purposes, however, there is a slightly more advanced form that is occasionally useful and is certainly worth knowing about. In the simple FOR loop each time through the loop one was added to the value of the index variable. This is sensible if you are using the index variable to count the number of times that the loop has been carried out. However, it is sometimes the case that a calculation carried out inside the loop needs a value that changes by something other than one each time through the loop. This is catered for by the addition of the BASIC statement STEP, the general form of which is:

FOR    'index        'start        'end  
      'variable' = 'value' TO 'value' STEP 'increment'

The 'increment' specified following STEP is the amount that is added to the index variable each time through the loop. So the simple FOR loop is equivalent to STEP 1. The only thing that you have to be careful of when using FOR...TO...STEP is making sure you know when the loop will come to an end. The rule is that the loop terminates when the value of the index *exceeds* the 'finish' value. So the value of an index variable in a FOR loop can never become larger than the 'finish value'. To see this in action try the following examples:

```
10 FOR A=.4 TO 10 STEP .01
20 PRINT A
30 NEXT A
```

and

```
10 FOR A=1 TO 100 STEP 25
20 PRINT A
30 NEXT A
```

The value of the *increment* following STEP can be negative, in which case it is better called a *decrement*. For example:

```
10 FOR A=100 TO 0 STEP -15
20 PRINT A
30 NEXT A
```

You may have some slight difficulty in working out when this and similar loops end. However, the rule is almost the same for a positive increment. Each time through the loop the value of the index variable decreases by the increment and the loop ends when its value first drops below the ‘end’ value.

As well as having a negative increment, it is possible for either the ‘start’ or the ‘end’ value to be negative and this is where things can be confusing. Consider this short program:

```
10 FOR A=-100 TO 50 STEP 10
20 PRINT A
30 NEXT A
```

At first sight it may not seem to make sense. However, if you keep a cool head then you should be able to see that the same rules apply. The value of the increment is added to the index variable each time through the loop until its value exceeds the ‘end’ value, if the increment is positive, or is less than the ‘end’ value, if the increment is negative.

## Using the FOR loop

There are one or two rules that govern the use of FOR loops. As you can use any valid BASIC statement within a FOR loop it is possible to use a GOTO or an IF to leave a FOR loop before it is finished. However this is to be avoided because your Oric will eventually get round to giving you an error message if you do it too often! Do not

leave FOR loops until they are finished! This rule also makes very good sense from the point of view of the logic of BASIC. If you want to leave a FOR loop before it is finished then this is a sure sign that you should have used a conditional loop (constructed using IF...THEN GOTO or REPEAT...UNTIL) rather than an enumeration loop. You could sum up the situation by saying 'never get an enumeration loop to do the work of a conditional loop and vice versa'!

It is often the case that FOR loops are used in combination. This is easy to understand as long as you think logically about the way each loop works. For example:

```
10 FOR A=1 TO 10
20 FOR B=1 TO 10
30 PRINT A,B
40 NEXT B
50 NEXT A
```

is correct because the FOR loop formed by lines 20, 30 and 40 is completely contained within the FOR loop starting at line 10 and ending at line 50. This means that the *inner* loop is carried out each time through the *outer* loop. It is all too easy to make a slight mistake and end up with:

```
10 FOR A=1 TO 10
20 FOR B=1 TO 10
30 PRINT A,B
40 NEXT A
50 NEXT B
```

which will give you an error message.

### **IF...THEN and the colon**

There is an even more general version of the IF statement than the one we have examined so far. As well as being able to follow the THEN by the BASIC command GOTO, you can in fact use any valid BASIC command. For example:

```
10 INPUT A
20 IF A<0 THEN PRINT "A is negative"
30 IF A=0 THEN PRINT "A is zero"
40 IF A>0 THEN PRINT "A is positive"
50 GOTO 10
```

The best way to think of this as a sort of easier version of the IF to skip an instruction. Only in this case the instruction is only carried out if the condition is true. There isn't very much to add to this description except to emphasise the fact that you can follow THEN by any valid BASIC statement – including another IF!

This extended version of the IF statement seems very useful at first but you quickly come to the conclusion that it's not often that you want to execute a *single* statement as a result of some condition. Clearly what is needed is some way of writing a list of BASIC statements following the THEN. Oric BASIC does provide a way of doing this although it is important to realise that not all versions of BASIC do. You can group together a number of BASIC statements on a single line, separating each one with a colon and they will be obeyed in turn from left to right. For example:

```
10 PRINT "First":A=1:PRINT "second";A
```

will be carried out as a single line of BASIC working from left to right and is equivalent to:

```
10 PRINT "First"
20 A=1
30 PRINT "Second";A
```

The use of colons to group a number of BASIC statements together effectively extends the rule for the default flow of control. Now instead of just obeying instructions in order of increasing line number (i.e. effectively moving down the screen) instructions that are grouped together on a line are obeyed from left to right. This is exactly the same way that a human would read a list of instructions from a sheet of paper (i.e. from left to right and then downwards). This ability to group instructions on a single line can be used to reduce the size of a program but if used too much results in it looking very dense and difficult to read. The best reason for grouping instructions together on a single line is that they actually have something to do with each other. For example if you are using two variables X and Y to keep track of the position of something on the screen, then it makes the program easier to read if, wherever possible, they appear on the same line. That is, instead of:

```
10 X=0
20 Y=0
```

use

```
10 X=0:Y=0
```

to show that X and Y are used together. However never crowd too many instructions together on a single line or it will make the program extremely difficult to read. Remember also that the Oric limits you to a line length of 78 characters (just over two screen widths) and it's always a good idea to leave some space to spare in case you want to add something extra.

As mentioned earlier, the place where it is really useful to have the ability to put more than one BASIC instruction on to a line is following a THEN. For example, at the end of most games programs it is usual to ask if the player would like another game. In the following example of a routine that could be added to the end of a program the answer is expected in the form of 1, to mean yes, or 0, to mean no. Any other value as a response will be taken to be a mistake and a message to that effect will then be printed and the question asked again.

```
110 INPUT "Do you want another game (Yes=1,No=0)";A
120 IF A=1 THEN GOTO xxx
130 IF A<>0 THEN PRINT "You must answer 1 or
    0":GOTO 110
140 PRINT "Bye bye!!"
```

Notice the use of the colon to group two BASIC statements together in line 130. The xxx in line 120 should of course be replaced by the line number of the start of the program. Being able to execute a group of BASIC statements following a THEN is very useful and it can make programs easier to write and easier to understand, but you will soon find that you cannot group very many statements together on one line before it becomes unreadable. There is another solution to this problem but for this we will have to wait until Chapter Five.

## IF...THEN...ELSE

As mentioned earlier, there is another special version of the IF statement that can be used to select which of two statements is carried out – the IF...THEN...ELSE statement. The general form of this is

```
IF 'condition' THEN 'list 1' ELSE 'list 2'
```

and its meaning is clear from the English meanings of the words THEN and ELSE. If the condition is true then 'list 1', a list of any

valid BASIC statements separated by colons, will be carried out. If the condition is false then 'list 2' will be carried out instead. For example:

```
IF A>=0 THEN PRINT "A is positive" ELSE PRINT "A is  
negative"
```

will print the first message if  $A \geq 0$  is true and the second message otherwise. You can use this form of the IF statement to replace the IF...THEN GOTO method of choosing which of two lists of instructions is to be obeyed. For instance the example involving reversing the sign of a number becomes

```
10 INPUT A  
20 IF A>=0 THEN B=A:PRINT "A is positive" ELSE  
   B=-A:PRINT "A is negative"  
30 PRINT A,B
```

But if the lists of instructions are more than a few statements long the IF...THEN...ELSE statement becomes quite unreadable! If this is the case then the earlier method of using IF...THEN GOTO is to be preferred even though its flow of control is a little tricky to follow at times. There is a way around this problem but once again we will have to wait for Chapter Five to find out what it is!

## END and STOP

There are two very simple BASIC statements that we have not discussed so far, that are very useful when used in conjunction with the IF statement. Suppose that as a result of some condition the program should stop, then currently the only way that we know of achieving this is to GOTO a line number that is at the end of the program. The BASIC statements END and STOP can be used anywhere in a program to return control to the user. For example:

```
10 PRINT "Do you want to continue Y=1,N=0"  
20 INPUT A  
30 IF A=0 THEN STOP  
40 IF A<>1 THEN GOTO 10  
50 PRINT "I continue master"  
60 GOTO 10
```

Notice the use of STOP in line 30. Also notice how line 40 is used so that if a value other than 1 or 0 is input the original question will be

repeated. If the user replies 1 then the program continues – in this case all it continues to do is to ask the same question until the user types 0 when it stops. The only difference between STOP and END is that STOP will report the line number at which the program halted, but END doesn't. In general a finished program should use END, but STOP is very useful while testing a program (see Chapter Ten).

### A final example

As a further example of both the IF and FOR statements consider the problem of turning the stopping distance program given in Chapter Two into a sort of quiz. For a range of speeds the player is asked for the thinking, braking and total stopping distance in feet.

```

10 PRINT "Stopping Distance Quiz"
20 MARK=0
30 FOR S=10 TO 80 STEP 10
40 PRINT "What is the thinking distance at ";S;" mph?";
50 INPUT T
60 IF T=S THEN MARK=MARK+1
70 PRINT "What is the braking distance at ";S;" mph?";
80 INPUT B
90 IF B=S*S/20 THEN MARK=MARK+1
100 INPUT "What is the total stopping distance?";D
110 IF D=S*S/20+S THEN MARK=MARK+1
120 NEXT S
130 PRINT
140 PRINT "You scored ";MARK
150 PRINT "out of a possible 24"

```

### The flow of control summarised

The GOTO statement can be used to transfer control to any point in a program, but to make programs easier to understand and therefore easier to debug, it is essential to use GOTO in ways that lead to simple changes in the flow of control. Its main use is in conjunction with the IF statement to form loops. The exit point from a conditional loop should either be placed at the beginning or at the end of a loop. The Oric provides the REPEAT ... UNTIL



statements to make conditional loops with exit points at the end particularly convenient. A second type of loop – the enumeration loop – can also be constructed using a special pair of instructions, FOR and NEXT. Apart from forming loops, the IF statement can also be used to skip sections of a program and to select between alternatives.

It is difficult to say exactly what goes on in the mind of an experienced programmer. Whatever it is, it is a process that marks the difference between a beginner and an expert. One thing that does seem certain is that, in addition to the catalogue of programs that have been seen before and a collection of handy tricks, the standard forms of the flow of control diagrams that we have been studying in this chapter are ever-present. It is worth mentioning at this point that it has been proved that you can write any program using only the default, select and conditional loop. This is so important that it is worth gathering together the flow of control diagrams so that you too can store them away in your personal memory.

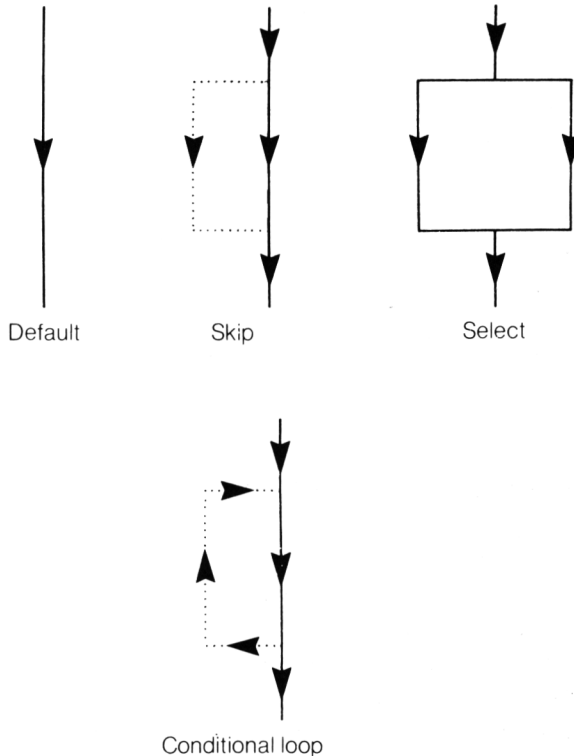


Fig. 3.7. Flow of control diagrams.

## Chapter Four

# Handling Text and Numbers

So far the only programs that we have written have used numbers. If this was all that computers could do they would be little different from pocket calculators! In this chapter we will look at how the Oric can handle characters and text just as easily as digits and numbers. In the last part of the chapter some other ways of extending the things we can do with data are introduced – arrays and tape storage.

### Strings

In Chapter Two the idea of a string – a collection of letters within double quotes – was introduced as a way of printing messages and prompts. In fact what we have been calling a string is really a *string constant*. The use of the word constant might alert you to the fact that there are such things as *string variables*. A string variable is similar to a simple variable in that it is a named area of memory that can be used to store information. In this case, the information is a collection of characters instead of a number. The rules for naming string variables are the same as for simple variables except that a string variable *must* end in a \$ sign. The dollar sign is used to distinguish between a simple variable and a string variable (e.g. 'B' is a simple variable but 'B\$' is a string variable and therefore different). The assignment statement can be used to store a string constant in a string variable and the PRINT statement can be used to print its contents.

```
10 A$="THIS IS A STRING"  
20 PRINT A$
```

In fact, a string variable can be used anywhere that a simple variable can, as long as it makes sense. For example, you can use INPUT A\$ to store a string typed in from the keyboard while a program is

running but `SUM=3+A$` is obviously nonsense (you cannot add a string to a number!). Notice in particular the difference between:

```
1Ø A$="1"
```

and

```
2Ø A$=1
```

Line 1Ø is fine because the 1 is enclosed in double quotes and is therefore a string, but line 2Ø will give an error message, “?TYPE MISMATCH ERROR IN 2Ø”, because A\$ is a string variable and 1 is a number.

The introduction of string variables is exciting because it opens up the possibility of the Oric handling text and even dialogues. So far, however, the only sort of program that we can write is:

```
1Ø INPUT "WHAT IS YOUR NAME ";N$
2Ø PRINT "HELLO ";N$;“, I AM YOUR ORIC
   COMPUTER”
```

which is all right for a start but it can result in unnatural dialogues like:

```
WHAT IS YOUR NAME ? FRED BLOGGS
HELLO FRED BLOGGS, I AM YOUR ORIC COMPUTER
```

The trouble is that although we can INPUT, PRINT and store strings we have no way of changing them. This is rather like being able to INPUT, PRINT and store numbers but having no way of doing arithmetic – it’s obvious that this would limit the programs that we could write! The answer lies in inventing an ‘arithmetic’ for strings so that as well as having arithmetic expressions we can use *string expressions*.

## String expressions

Before introducing the Oric’s facilities for handling strings it is worth considering what sort of things you might like to do and then see if the Oric can actually fit the bill. If a program had someone’s first name in F\$ and their last name in L\$ then it would be useful to be able to *join* them together to form one longer string. Such joining together of strings is known as *concatenation*. Another thing that would be useful is the ability to *extract* part of a string. For example, you could extract the last name from a string consisting of an initial

and surname, i.e. extract "BLOGGS" from "F.BLOGGS". A string that is part of another string is often called a *substring*. For example, BLOGGS is a substring of F.BLOGGS. Another useful facility would be to replace a substring by another. For example, if we were trying to keep F.BLOGGS a secret we might want to replace the surname by asterisks giving "F.\*\*\*\*\*". Finally it would be a great advantage to be able to test for the presence of a particular substring in a string, for example, to see if the substring "BLOGGS" occurred in the name stored in N\$. To recap, the string operations that we would like to find are – concatenation, substring extraction, substring replacement and substring searching.

Our first requirement, string concatenation, is immediately satisfied by the Oric's concatenation sign, +. If A\$ contains the string "ABCD" and B\$ contains "EFGH" then after:

```
C$=A$+B$
```

C\$ contains "ABCDEFGH". Notice that we now have two uses for the symbol +, as the sign for addition and as the sign for concatenation. You can use the + sign more than once in a string expression:

```
C$="MR "+F$+" "+L$
```

will join up the four strings involved and if F\$ contains a first name "FRED" and L\$ a last name "BLOGGS" then C\$ will contain "Mr FRED BLOGGS". Notice the use of a single space between the two strings to avoid the result being "MR FREDBLOGGS"!

Extracting or changing a substring can be done by using a single new operation – MID\$. If 's' is a string of characters:

```
MID$(s,m,n)
```

is the substring consisting of 'n' letters starting with letter 'm'. For example:

```
PRINT MID$("12345678",3,4)
```

will print "3456", i.e. the substring consisting of four characters starting at the third character. The MID\$ operation is easy to understand and use as long as you remember that the first number indicates where the substring should start and the second number indicates how many characters should be in the substring.

You can use arithmetic expressions to specify 'm' and 'n', for example:

MID\$("ABDC",FIRST,2)

and

MID\$("ABDC",I,L+3)

are both valid and 'FIRST', 'I' and 'L+3' are used in place of constants. However, by now you may have realised that one of the most powerful principles in BASIC is that anywhere you can use a constant or a variable you are also allowed to use an expression. You shouldn't be frightened to write complicated string expressions any more than you would be over complicated arithmetic expressions.

You can't really make a mistake using MID\$. If you specify that the substring should be longer than the number of letters to the left of the starting position then you will just get as many letters as there are! For example MID\$("12345",4,10) is just "45". If you leave 'n' out then the substring will consist of all the letters to the end of the string 's'. For example MID\$("12345",4) is also "45". The only error that you can make is to specify a starting position less than 1. For example MID\$("12345",0,10) will give you the error message, "?ILLEGAL QUANTITY ERROR". However if you use a correct starting position and specify a length of zero you will not get an error but a substring with no characters in it! This is a very special string – the *null string*. The null string has no characters in it and plays a very similar role in string expressions to that of zero in arithmetic expressions. The string constant that corresponds to the null string is written "", i.e. a pair of double quotes with nothing in between. Notice the difference between "" and " ". The first is the null string and has no characters in it, the second is a string consisting of one character – a blank or space. From the point of view of a computer, a space is just as much a character as a letter or the alphabet, it takes up one position to print and it needs just as much computer memory to store! If you print a null string it has no effect whatsoever. The statements PRINT A;B and PRINT A;"";B produce the same result.

As an example of using MID\$ consider the problem of printing the name of a month of the year given its number (i.e. you should print DEC for 12 and MAY for 5 etc.). Try the following short program:

```
10 Y$="JANFEBMARAPRMAYJUNJULAUGSEPOCT
   NOVDEC"
20 INPUT "MONTH NUMBER ";MN
30 PRINT "MONTH ";MN;" IS ";MID$(Y$,1+3*(MN-1),3)
40 GOTO 20
```

If you enter a number in the range 1 to 12 the program will print the correct abbreviation for the month in question. The way that it works is by extracting the three letter name from the long string Y\$. The best way to understand the use of MID\$ in line 30 is by working it out by hand for a few values of 'MN'. If 'MN' is 4 then  $(1+3*(MN-1))$  is 10 and the substring is given by MID\$(Y\$,10,3) which gives the three letters "APR".

The MID\$ operation will extract any substring from a string and in this sense it is all you need. However, there are two other ways of extracting substrings that have become common and indeed popular in BASIC – LEFT\$ and RIGHT\$. LEFT\$(s,n) will extract the string with 'n' characters in it starting from the first character in 's'. In other words the leftmost 'n' characters. You should be able to work out that LEFT\$(s,n) is the same as MID\$(s,1,n). RIGHT\$(s,n) will extract the last or rightmost 'n' characters from the string 's'. You may think that this has no equivalent MID\$ operation but once you know that the Oric will 'tell' you the length of a string on request the equivalent is obvious. If you write LEN(s) the Oric will count the number of characters in a string. For example PRINT LEN("ABCD") will print 4 on the screen. Now you should be able to work out that RIGHT\$(s,n) is the same as MID\$(s,LEN(s)-n,n).

There is no way of directly changing characters that are part of a string. In other word if S\$ contains "AB3D" there is no way of directly altering the "3" to "C". However, using MID\$ you can take the string apart and insert the new letters. For example to replace the 3 in S\$ with C you could use:

```
S$=MID$(S$,1,2)+"C"+MID$(S$,4,1)
```

The first MID\$ extracts "AB" and the second the final "D". Putting them all together gives "ABCD" as required. This is the only way that you can alter part of a string.

The only thing left from our initial list of string handling requirements is testing to see if a particular substring is present in another string. The Oric's version of BASIC doesn't provide a direct method of doing this but it does extend the use of conditional expressions (see Chapter Three) to strings and this can be used to achieve the same ends.

You can use all of the relations that were introduced in Chapter Three with strings. The meaning of = and <> are easy enough to understand. Namely, two strings are equal if they are of the same length and contain the same characters in the same order otherwise they are not equal. However, what do the relations <, >, <= and

$>=$  mean when applied to strings? The answer to this question varies from BASIC to BASIC. In the case of the Oric, however, they are defined so that  $A\$ < B\$$  is true in the string in  $A\$$  would come before the string in  $B\$$  in an alphabetically ordered list of strings. The trouble is that we are all so familiar with alphabetically ordered lists that we tend to forget how they work! If the two strings being compared are single letters, then  $A\$ < B\$$  if the letter in  $A\$$  comes earlier in the alphabet than the letter in  $B\$$ . For example, “A” < “B” is true but “D” < “B” is false. What about comparing strings that contain single characters that are not necessarily letters, i.e. what do we make of “\*” < “\$”? In the case of the letters, the alphabet provided us with a ready-made order so what we need is to extend this order to include all the other symbols that the Oric can use. In other words, we need a *super alphabet*! This is already available for the Oric. If you look at Appendix D in the Oric Manual you will see a listing of the complete set of Oric characters in a predefined order and it is this that is used as the super alphabet to decide if  $A\$ < B\$$ . If you don't have the Oric Manual to hand, or if you are just interested, you can print all the characters in their proper order by using the following program:

```
10 FOR I=32 TO 126
20 PRINT "CHARACTER ";I;" = ";CHR$(I)
30 NEXT I
```

Line 10 specifies 32 as the starting point as the characters before that are unprintable. Notice that character 32 itself is a space and actual characters start with 33. Don't worry about the use of  $CHR\$$  in line 20, this will be explained later. Coming back to the question of whether or not “\*” < “\$” is true or false, “\*” is character 42 and “\$” is character 36 so “\$” comes before “\*” in the order and “\*” < “\$” is false. You can decide the truth or otherwise of any relation in the same way.

If the two strings contain more than one character they are compared one character at a time until the first pair of different characters is found. The relationship between the two string is then decided on the basis of those two characters. For example,  $ABCD < AZCD$  is true because the first pair of letters that are different is B and Z and  $B < Z$  is true. If one of the strings is the same as the other apart from the addition of a few extra characters then the comparison is based on length, i.e.  $ABCD < ABCDEF$ , because there is no pair of letters that is different and  $ABCD$  is shorter than  $ABCDEF$ .

Now we can do everything that we wanted to with strings! However there are other string operations and these will be introduced in the next chapter.

### **Another type of number – integers**

Oric BASIC recognises two different types of number – *real* and *integer*. The type of numeric constant and variable that we have been using so far are more properly called real constants and real variables. The difference between the two types of number is that a real number can have a fractional part but an integer cannot. So for example, 3.1 is real because it has a fractional part but 3 is an integer. You may be puzzled about the nature of the number 3.0. It has a fractional part but it is zero, so is it real or integer? The answer is that it is real, because the fact that .0 has been included can only be taken to imply that the number could have a fractional part but in this case it just happened to be zero. The idea that a real number is allowed to have a fractional part while an integer isn't is perhaps easier to see when it comes to real and integer variables. An integer variable is distinguished from a real variable by ending in %. So NUM is a real variable and can store numbers with fractional parts but NUM% is an integer variable and can only store whole numbers. To see this try:

```
1Ø NUM=12.345
2Ø NUM%=12.345
3Ø PRINT NUM,NUM%
```

It is quite possible to ignore the existence of integers and treat all numbers as real. However there are certain advantages to be gained from using integer variables. In particular:

Integers can take less memory to store.

Arithmetic is faster with integers.

Integers do not suffer from rounding errors.

On the other hand, real variables are easier to use in that you don't have to think about the type of number that is going to be stored in a real variable. Real numbers can also store numbers over a much greater range than integers.

Whether you choose to take advantage of integer variables is up to you. In general unless speed or absolute accuracy is important you might as well use real variables. Indeed many dialects of BASIC and



more advanced languages make no distinction between integers and reals! There is another point of view that maintains that a real variable should only be used when it is absolutely necessary, because this forces the programmer to think very carefully about what any given variable will be used for. What seems to be important is that where a language gives you the choice of real and integer variables you should be able to recognise the distinction between them, and so be in a position to make up your own mind when to use one or the other!

## Arrays

Strings and numbers are the only two types of data that the Oric can handle and this is quite sufficient for most purposes. However, the Oric does provide a way of using the basic types of data in a more sophisticated way, the *array*.

Consider the problem of reading in five numbers and printing them out on the screen in the reverse order. So far the only method that we could use is:

```
10 INPUT A1,A2,A3,A4,A5
20 PRINT A5,A4,A3,A2,A1
```

which is not too bad for five numbers but think what the program would look like if the problem was to reverse 100 numbers!

What we need to be able to do is to refer to a variable like 'A(I)' where I can take values from 1 to 5 in a FOR loop. Then we could write:

```
20 FOR I=1 TO 5
30 INPUT A(I)
40 NEXT I
50 FOR I=5 TO 1 STEP -1
60 PRINT A(I)
70 NEXT I
```

This is in fact exactly what BASIC allows you to do. The collection of variables A(1) to A(5) is called *the array A* and a particular variable A(I) is called an *element* of the array (see Fig. 4.1). The only complication is that before you can use an array you must tell your Oric how many elements the array is going to have. This is done using the DIMension statement:

10 DIM A(5)

which should be added to the previous program to make it work! If you define a variable as having only five elements and then try to use A(6) you will get an error message for trying to use something that doesn't exist! It is tempting to think that it is better to define arrays larger than you need to try to avoid such error messages but be warned, arrays can quickly use up all the memory that your machine has to offer!

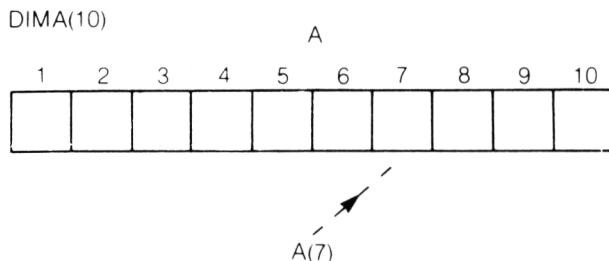


Fig. 4.1. A one-dimensional array with 10 elements.

There are two slight complications that haven't been mentioned so far. The first is that if you use DIM A(10) then the Oric will actually create an array with 11 elements. Where you might ask is the extra element? The answer is that every array has a zeroth element. For example, DIM A(10) creates A(0) to A(10) i.e. eleven elements! The reason why this extra element has been ignored until now is that many versions of BASIC do not provide zeroth elements in arrays. If you fall into the habit of using elements like A(0) then your programs won't work on BASICs that do not provide A(0) but if you avoid using any zeroth element the only consequence is that you waste a tiny amount of memory needed to create A(0). For this reason the zeroth element of every array will be ignored! The second problem is that you can only dimension an array once in a program! If you first use DIM A(10) and then later use DIM A(20) or even DIM A(10) again you will get an error message – so keep DIM statements out of loops!

In addition to being able to define arrays that can be thought of as *rows* of variables (see Fig. 4.1) you can define arrays that correspond to organising variables into tables made up of rows and columns (see Fig. 4.2). For example,

DIM A(10,10)

defines a collection of variables organised into 10 rows and 10 columns. A particular element of this array can be referred to as

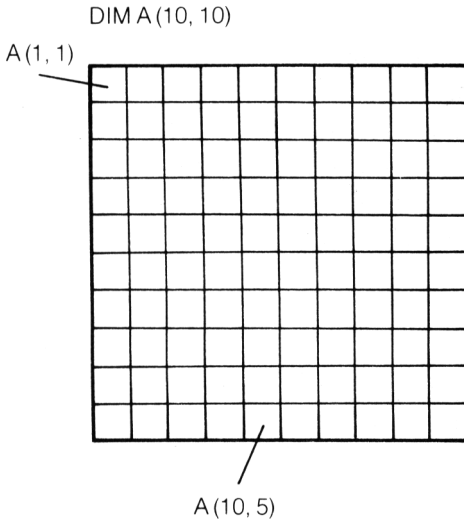


Fig. 4.2. A two-dimensional array.

$A(I, J)$  where the two indices select the row and column.

Although in theory the idea of a two-dimensional array can be extended to three-, four- and  $n$ -dimensions, you will find that you run out of memory very rapidly if you go beyond two dimensions.

You can form arrays from integers and string variables as well as real variables. For example `DIM NUM%(10)` is an integer array of 10 elements. Integer arrays should be used in preference to real arrays wherever possible because they use up much less memory. String arrays are particularly important can be used to store and manipulate lists of words. For example,

```
DIM S$(10)
```

is a string array composed of 10 different strings. Each element of a string array behaves exactly like a standard string and can be used to store a variable number of characters. For example, the number reversing program can be used to reverse a list of words:

```
10 DIM A$(5)
20 FOR I=1 TO 5
30 INPUT A$(I)
40 NEXT I
50 FOR I=5 TO 1 STEP -1
60 PRINT A$(I)
70 NEXT I
```

You can use MID\$, LEFT\$, etc., and any other string operations on any element of a string array.

## A word game

As an example of using string arrays consider the problem of writing a program to play the game of hangman. Because the computer cannot 'think up' a list of words for you to guess it is necessary to ask someone else to type in a list for you to guess. Once the list of words has been entered the player must try to guess each word in turn letter by letter. Each letter that is entered must be checked against each letter in the word. If it is present then the letter in the word must be replaced by a blank to make sure that the player cannot guess it a second time. When all the letters have been guessed the program moves on to the next word, or if there is none, comes to an end. After this description you should be able to make a good attempt at your own hangman program before looking at the one below:

```

10 DIM W$(5)
20 FOR I=1 TO 5
30 INPUT "WORD = ";W$(I)
40 NEXT I
50 FOR I=1 TO 5
60 G=0
70 T$=W$(I)
80 G=G+1
90 INPUT "GUESS=";A$
100 F=0
110 FOR J=1 TO LEN(W$(I))
120 IF LEFT$(A$,1)<>MID$(W$(I),J,1) THEN GOTO 160
130 PRINT "YES !-";LEFT$(A$,1)
140 W$(I)=MID$(W$(I),1,J-1)+" "+MID$(W$(I),J+1)
150 A$=""
160 IF MID$(W$(I),J,1)<>" " THEN F=1
170 NEXT J
180 IF F=1 THEN GOTO 80
190 PRINT "YOU GOT IT IN ";G;" !!"
200 PRINT "THE WORD WAS ";T$
210 NEXT I
220 PRINT "GAME OVER"

```

Line 10 defines the array W\$ so that it can hold five words. Lines 20

to 40 are used to input the words. The FOR loop starting at line 50 and ending at line 210 repeats the guessing part of the program five times, once for each word. The variable T\$ is used in line 70 to store the word until the end of the game so that the array element can be modified by storing blanks in the place of letters that have been guessed. The guess is input in line 90. Each letter in the word is checked against the current guess by the FOR loop starting at line 110 to line 170. The IF statement at line 120 is used to skip the lines that deal with what happens if a letter is guessed correctly. The variable 'F' is used to check that there are still letters left to be guessed. Variables such as F that are used to record a result of part of the program for a later part to test are usually called *flags*.

In this one program you can find examples of the FOR loop, the conditional loop and the IF skip as described in Chapter Three. See if you can find the lines that make up each one.

### Initialising variables - DATA and RESTORE

It often happens that a standard set of values needs to be stored in a set of variables or an array for a program to work properly. For example, suppose we want to print the number of days in a given month we could set up an array of 12 elements and *store* the answer in each element – i.e. the number of days in Jan would be stored in the first element of the array, the number in Feb in the second and so on. This is a very simple and useful idea but how do you initialise each element of the array to the correct value? You could use a FOR loop and an INPUT statement to read in the answers or you could write 12 assignment statements. To make life slightly easier, BASIC provides the facility to store data within a program so that it can be transferred to any variables of your choice. The data is stored in a DATA statement that is composed simply of the word DATA followed by a list of values separated by commas. For example, the days in each month could be written as:

```
10 DATA 31,28,31,30,31,30,31,31,30,31,30,31
```

To transfer these data values into variables the READ statement is used. A READ statement is simply the word READ followed by a list of variables separated by commas. Each time a READ statement is encountered data values are transferred into variables in the variable list – one data value per variable. The best way to think of this is to imagine a pointer initially set to the first data value in the

DATA statement. Each time a READ statement transfers a data value into a variable, the pointer is moved on to the next data value. Whenever a READ statement is obeyed data is transferred, starting from whatever data value it has reached after previous READ statements. So the array holding the 12 data values in the example DATA statement given above can be produced using:

```
20 DIM M(12)
30 FOR I=1 TO 12
40 READ M(I)
50 NEXT I
```

You can have as many DATA statements in a program as you like, anywhere that you like, and they are treated as if all the data that they store was contained in one big DATA statement. So when the pointer moves past the end of a DATA statement it moves to the beginning of the next DATA statement. If there isn't one, however, you'll get an error message.

There is no restriction on the type of data that you can store in a DATA statement – i.e. you can use strings or numbers – but you must always be careful to READ data into variables of the same type – i.e. strings into string variables and numbers into numeric variables. Strings in DATA statements do not have to have double quotes around them unless they include commas – otherwise it wouldn't be possible to tell where one data item ended and another began. As an example of using strings in DATA statements consider the following:

```
10 DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,
    SEP,OCT,NOV,DEC
20 DIM M$(12)
30 FOR I=1 TO 12
40 READ M$(I)
50 NEXT I
```

and compare it to the example given earlier.

Sometimes, especially in games, it would make things easier if we could re-read the data. This can be done using the RESTORE command. Following RESTORE that pointer is back at the beginning of the first DATA statement in the program. For example try:

```
10 DATA 1,2,3,4
20 READ, A,B,C,D
```

```
30 PRINT A,B,C,D  
40 RESTORE  
50 GOTO 20
```

which will read the same four numbers over and over again.

### **Tape storage**

Most machines provide commands that will allow data – that is the contents of simple variables or arrays – to be stored on tape in the same way that programs can using CSAVE. However the Oric doesn't provide any tape handling facilities as standard and there is nothing that can be done about this by the BASIC programmer. There are plans to produce special software for data handling but without this programming extensive data using the Oric is difficult to say the least.

## Chapter Five

# Functions and Subroutines

At this point in learning BASIC you should be in a position to see that the expression is the main way that programs *change* data. Without expressions – arithmetic, conditional and string – BASIC would be reduced to moving values from one place to another. It is only by the use of expressions that values can be combined and compared to produce new results. To make expressions even more useful, BASIC provides a large range of operations that can be used to make expressions, in the form of *functions*. You can also extend the range of functions by creating your own, *user-defined* functions. This creation of new operations can be taken one stage further in BASIC by using the GOSUB and RETURN statements to group statements together into *functional units* or *subroutines*.

In the first part of this chapter we look at the general idea of a function and then move on to examine some of the more common functions available to the Oric programmer. The sections that deal with particular functions can be read very quickly or even skipped until you need to use them or until they are used in an example. However, don't skip the section on *special functions* because these are particularly important.

### The idea of a function

Before dealing with the way the Oric handles functions, it is worth looking at functions in general. You may already be familiar with the idea of a function from mathematics. For example,  $\sin(x)$  is a function. However, the idea of a function isn't really anything to do with advanced mathematics. At its most simple, a function is an operation on data that produces a *single* value as its result. For example, finding the larger of two numbers is an operation on data that returns a single value – the maximum of 3 and 42 is 42, the



maximum of 2 and 2 is 2 – and *maximum* is therefore a function.

The Oric doesn't have a maximum function but it is nevertheless a useful one to consider as an example because it is easy to understand.

The standard way of writing a function involves writing its name to the left of the values on which it is to operate that are enclosed in brackets. In the case of finding the maximum of two numbers a sensible name for this function is 'max' and so the previous two examples can be written:

```
max (3,42)
and
max(2,-2)
```

Following the usual BASIC convention that anywhere that you can use a constant or a variable you can use an expression, the following is also allowed:

```
max(count+3,total*20)
```

The data values that follow the name of the function are called *parameters*. It is possible for functions to have any number of parameters but the functions supplied on the Oric use one, two or at most three parameters.

Functions can be used in expressions just as if they were variables or constants. For example,

```
RESULT=max(3.3,4.2)
```

would evaluate our function 'max' and store the answer (4.2) in 'RESULT'. (Remember that the Oric doesn't have a 'max' function so don't try this example.) You can now see why the condition that a function should give only one result is so important. If a function gave more than one result, which one would be stored in the variable 'RESULT' or which one would be used to evaluate the rest of the expression? As we want to use functions in expressions they *can* only return one answer. Sometimes it is possible to change something that isn't a function into a function by simply choosing one of the possible answers. For example, the Oric has a function SQR which gives the square root of a number. Now if you ask for the square root of four the answer is obviously two i.e. two times two is four, but it is all too easy to forget that minus two is also the square root of four. A minus times a minus is a positive and so  $-2*-2$  is 4 (not -4). The objection that the square root operation isn't a function can be avoided by simply deciding that SQR will be a function that returns the positive square root of a number.

## The Oric's functions

The Oric has a very wide range of functions and some of them are so specialised that it is better to deal with them in detail in other chapters. However, there is a *central core* of functions that you would expect to find in any BASIC and these will be explained in this chapter. A full list, with brief explanations, of all the Oric's functions can be found in its manual.

The *core* functions can be divided into three groups: the arithmetic functions such as SQR and ABS; the trigonometrical functions such as SIN and COS; and the string functions such as LEN and CHR\$. In addition there are a number of unclassifiable but very important *one-off* functions such as RND. The most important thing is to have some idea of what functions are available, so a brief reading of the description of each function listed below is recommended. However, it is difficult to appreciate, let alone remember, the subtler details of the use of a function until you actually *need* to use it! If you want to see the effect of any of the functions use the following program:

```
10 INPUT X
20 PRINT SIN(X)
30 GOTO 10
```

but change the PRINT SIN(X) to the function that you are interested in.

## Arithmetic Functions

### *ABS – ABSolute value of a number*

The absolute value of a number is obtained by ignoring its sign and treating it as positive, i.e. ABS(−2) is 2 and ABS(2) is 2.

### *EXP – EXPOnential function*

EXP(X) raises 'e', the exponential number, which has the value 2.718281, to the power of 'x'. That is, EXP(X) is the same as  $e^x$ .

It is difficult to explain why this function is so important but it crops up in just about every area of mathematics, (see also the function LOG). When using EXP it is well worth being aware of the fact that EXP(X) gets very big for even small values of 'X'. The largest integer that EXP(X) can accommodate on the Oric is 88. Larger numbers will cause an "OVERFLOW ERROR".

*INT – round a number down*

The INT function will remove the fractional part of any number and so convert it into an integer (see Chapter Four). This is achieved by rounding the number *down* to the nearest whole number. For positive numbers this corresponds to *chopping off* the fractional part, e.g. INT(3.21) is 3. For negative numbers things are a little more complicated. Rounding a negative number down looks a little strange, e.g. INT(-4.7) is -5. But this is simply because  $-4 > -5$  i.e. -5 is *smaller* than -4!

*LN – Natural Logarithm of a number*

The natural logarithm of a number is the power that you have to raise 'e' to give the number. Most people are more familiar with a slightly different form of the logarithm. The logarithm that is given in most log tables is in fact the logarithm to the base 10. In other words it is the number to which 10 has to be raised to give the original number. Oric BASIC includes both versions of the function. As LN(X) is the number to which you have to raise 'e' to get X you should be able to see that EXP(LN(X)) is X.

*LOG – the log to the base 10*

LOG(X) is the number to which you have to raise 10 to get X. This function has already been partially described in the last section. You must be careful not to confuse LN and LOG.

*PI ( $\pi$ )*

This is a very odd function in that it has no parameters and always returns the same result ( $\pi=3.14159265$ ) but it is very useful. As everyone knows, the area of a circle is given by  $\pi r^2$  and this translates to BASIC as:

```
10 AREA=PI*R*R
```

*SGN – the SiGN of a number*

The sign of a number is +1 if the number is positive, -1 if it is negative and 0 if it is zero. For example, SGN(-232) is -1 and SGN(3239) is 1.

*SQR – the SQuare Root of a number*

The square root of a number is a result that when multiplied by itself gives the original number i.e.:

SQR(X)\*SQR(X) equals X

Notice that negative numbers do not have square roots because if you multiply any number, even a negative one, by itself you will get a positive number. If you try to take the square root of a negative number you will therefore get an “ILLEGAL QUANTITY ERROR”.

## The trigonometrical functions

The best known examples of functions are probably the *trigonometrical* or *trig* functions. It is beyond the scope of this book to go into detail about the theory of trigonometry and Chapter 7 of the Oric Manual covers the definitions of TAN, SIN and COS. There is one use for the trig functions that is important to nearly every computer user interested in graphics, namely drawing circles. Again Oric users are particularly lucky, in that Oric BASIC includes a command that will plot a circle (covered in Chapter Eight) and its presence avoids the need to go into the trigonometrical details.

If you do need to use any of the trig functions, it is important to realise that the Oric doesn't measure angles in degrees but in radians. If you want to convert an angle in degrees to radians then use:

$$\text{radians} = \text{degrees} * \text{PI} / 180$$

Where PI is approximately 3.14159. To convert radians to degrees use:

$$\text{degrees} = \text{radians} * 180 / \text{PI}$$

The three trig functions available on the Oric are:

SIN = sine of an angle measure in radians  
 COS – cosine of an angle measured in radians  
 TAN – tangent of an angle measured in radians

The Oric also has available the inverse function related to the TAN function.

### *ATN – ArcTanGent*

The arctangent of a number is the angle in radians whose TAN is equal to the number, i.e.  $X = \text{TAN}(\text{ATN}(X))$ .

If you need the inverse of SIN or COS, or any of a number of other *derived trigonometrical functions* you will find details of how to define them as *user-defined functions* (see later in this chapter) in the Oric Manual (Appendix G).

## String Functions

We have already met most of the string functions in Chapter Four. For completeness, and to provide a handy reference, all of the string functions are described here. However only the new functions are described in any detail. To see how the functions that require string input work, modify the following program:

```
1Ø INPUT X$
2Ø PRINT ASC(X$)
3Ø GOTO 1Ø
```

### *ASC – the ASCII code of a character*

The ASC function does the opposite to the CHR\$ function in that it returns the position in the list of characters of any particular character. For example, ASC("A") is 65 and CHR\$(65) is "A". If ASC is applied to a string of more than one letter, the code of the first character in the string is returned as the result. If the string is null, i.e. contains no characters, the code returned is zero.

### *CHR\$ – CHaRacter function*

CHR\$(N) will give the character that is at the Nth position in the list of all the Oric's characters. CHR\$ will return all the characters that the Oric can use even if they cannot be printed on the screen. So, if you type CHR\$(8), or any number up to and including 33, all you will see is a blank screen.

### *HEX\$ – HEXadecimal representation*

HEX\$ is a function that you certainly won't use unless you are going to become involved in the technicalities of how the Oric works! Its action is very simple to describe: it converts any number into a string of characters that represents the number in base sixteen. In other words, it returns the hexadecimal representation of the number. For example, HEX\$(15) is "F" and HEX\$(255) is "FF". Before you can use HEX\$, however, you need to understand something about hexadecimal numbers.

### *LEFT\$ – the left part of a string*

LEFT\$(string,N) returns the first N characters of a string.

### *LEN – the LENgth of a string*

The LEN function returns the length of any string. For example, LEN("COMPUTER") is eight and LEN("") (the null string) is 0.

*MID\$ – the mid part of a string*

MID\$(‘string’,M,N) returns a string of length N starting at character M. If N is omitted then the entire string to the right of character M is returned.

*RIGHT\$ – the right part of a string*

RIGHT\$(‘string’,N) returns the last N characters of a string.

*STR\$ – converts numbers to strings*

The STR\$ is a function that is useful for advanced applications. It converts any number (or the result of an expression) to the string of characters that would be displayed if the number (or the result of the expression) were printed. The STR\$ function therefore provides a link between numbers and strings – for example, “JULY ”+STR\$(31) works out to the string “JULY 31”. There is a bug in early versions of the Oric’s BASIC ROM that causes control characters to be embedded in the result of STR\$ and these change the colour that the resulting string is displayed in. This problem is not difficult to solve and is described in more detail in the next chapter.

*VAL – converts digit strings to numbers*

The VAL function is the opposite of the STR\$ function in that it converts a string into a number. The string can be any sensible collection of digits, i.e. anything that you could type in response to an INPUT statement. For example VAL(“34”) is 34 and if A\$=“123.456” then VAL(A\$) is 123.456.

## **Special functions**

There are three functions, RND, KEY\$, GET and FRE that are so generally useful that it seems worthwhile to treat them on their own and at some length. RND and FRE are functions that return numbers so they could otherwise have been treated in the section on arithmetic functions. KEY\$ and GET return a character so they could otherwise have been treated as a string functions.

### *RND*

RND(1) is a function that returns a number in the range 0 to less than 1, which can be treated as if it were random. To say that a computer can give a number at random always sounds like a contradiction and indeed to some extent it is. The point of confusion

comes from the use of the word *random*. If you are using the computer to play a game then all that you need is a sequence of numbers that are not predictable by anyone playing the game. In other words, for most purposes a list of numbers can be said to be random if there is no detectable pattern. If you run the following program:

```
10 PRINT RND(1)
20 GOTO 10
```

you should see a list of numbers that shows no obvious pattern. (In fact there is a pattern but it is so complicated it takes a *Oric* to follow it!) This sort of randomness is more correctly called *pseudo randomness* and the RND function is a *pseudo random number generator*. The numbers that it produces are *evenly spread* throughout the range 0 to less than 1, i.e. any number is just as likely to 'come up' as any other and there should be no discernible pattern that would help someone predict the next number that RND will produce.

The main trouble with RND is that it's not often that we need a random number in the range 0 to less than 1. We normally need the program to do one of a number of different things at random. The best way of doing this is to change the RND into a random whole number between 1 and N where N is the number of anything we want to select from, using the formula:

$$\text{INT}(\text{RND}(1)*N)+1$$

For example if you want to program a six-sided dice or choose at random between six different things, the value of N would be 6 and:

```
10 PRINT INT(RND(1)*6)+1
20 GOTO 10
```

will print numbers from 1 to 6 with approximately the same frequency and in such a way that there should be no obvious pattern. The subject of how to use random numbers in programs is too vast to cover in this book but many examples will crop up in other chapters. If you want to explore randomness further, Chapter Five of *The Complete Programmer* by Mike James (published by Granada) is devoted to this topic.

### KEY\$ and GET

The function KEY\$ is closely related to INPUT in that it can be used to *read in* a single character from the keyboard. The difference is

that INPUT A\$ waits for something to be typed on the keyboard until the RETURN key is pressed, but KEY\$ doesn't wait. If you try the following program:

```
10 INPUT A$
20 IF A$<>"" THEN PRINT A$
30 GOTO 10
```

you will have to press RETURN before you see anything on the screen. However, if you change line 10 to:

```
10 A$=KEY$
```

the character corresponding to any key that you press appears on the screen at once. Notice that another difference between INPUT and KEY\$ is that KEY\$ doesn't automatically print anything that you type on the screen.

Whenever the Oric meets the KEY\$ function it immediately examines the keyboard. If there is a key already pressed the appropriate character is returned by the function. If no key is pressed the function returns the null string. No matter what has happened the KEY\$ function does *not* wait for a key to be pressed.

The main use of KEY\$ is in games where the *arrow* keys are used to control the movement of something on the screen. The following program indicates how the Oric detects that one of the arrow keys has been pressed:

```
10 A$=KEY$
20 IF A$="" THEN GOTO 10
30 IF A$=CHR$(8) THEN PRINT "LEFT"
40 IF A$=CHR$(10) THEN PRINT "DOWN"
50 IF A$=CHR$(11) THEN PRINT "UP"
60 IF A$=CHR$(9) THEN PRINT "RIGHT"
70 GOTO 10
```

Line 10 gets the character corresponding to any key pressed on the keyboard, if any. Line 20 tests to see if A\$ is the null string, i.e. no key has been pressed, and if it is, sends control back to line 10. Thus the loop formed by line 10 and 20 only stops when a key is pressed. Then lines 30 to 60 test to find out which of the arrow keys are pressed and print an appropriate message. As three of the arrow keys produce unprintable characters there is no choice but to use CHR\$ functions with the correct character codes within the IF statements. Line 70 repeats the whole program. Notice that if any key other than an arrow key is pressed the loop formed by lines 10



and 20 stops but nothing is printed on the screen. In Chapter Eight an example is given where the same sort of program is used to drive a dot around the screen.

The command GET isn't written in the usual style of a function but it still returns a value.

GET A\$

works in much the same way as A\$=KEY\$ in that the keyboard is examined and if a key is pressed the corresponding character is stored in A\$. The difference is that GET will wait until a key is pressed. For example, you can delete line 20 from the previous program if you replace line 10 by:

10 GET A\$

because GET automatically waits until a key is pressed and so it cannot return the null string.

### *FRE*

The function FRE(0) will return as a result the amount of memory that is unused and hence free. Thus to find out how much memory you have left type:

PRINT FRE(0)

You can also use FRE to test whether your program is running on a 16K or 48K Oric or whether there is room to dimension a particular array.

This use of FRE to find out how much memory is unused is simple enough. However, there is an additional way of using FRE that is far from obvious. During string operations memory is used up to hold temporary results and copies of old versions of the strings involved. This memory is not immediately returned to the unused portion of RAM and so you can be in the position of storing rather more data than you need. Although the Oric will eventually get round to clearing up the memory and removing such *garbage* as unwanted string values, leaving this until it is necessary can make the task take some time. However you can force the Oric to perform *garbage collection* by using:

A=FRE("")

Where the A can be any variable that you are not using. In general, unless you are writing very large programs or going in for a lot of string handling, don't worry about using FRE("").

**User-defined functions – DEF FN and FN**

In our general introduction to functions, the idea of a function to find the maximum value of two numbers was discussed but it was pointed out that, although the Oric has a wide range of functions, such a 'max' function isn't among them. Oric BASIC does allow definition of new functions but not with enough freedom to allow us to create such a function to determine the maximum of two numbers. However even this limited ability is worth having.

You can define a new function in terms of an expression. For example, the Oric lacks a *square* function, i.e. one that will work out the square of any number. The names of all user-defined functions are 'FN' followed by one letter, so you can define the new function 'FNS' to fill this gap by:

```
DEF FNS(X)=X*X
```

The word DEF is used to indicate that this is a function definition. The meaning of this function should be clear – whenever the Oric sees the function called 'FNS' it squares the parameter within the brackets next to it. For example:

```
A=FNS(2)
```

would result in 4 being stored in the variable 'A'. This is all fairly straightforward but what about the following:

```
10 DEF FNS(X)=X*X
20 X=3
30 Z=4
40 A=FNS(Z)
50 PRINT "X =";X;" Z =";Z;" A =";A
```

Before you run this program try to work out what the values of 'X', 'Z' and 'A' will be. The correct answer is that 'X' is 3, 'Z' is 4 and 'A' is 16. Any possible confusion comes from the fact that 'X' is used in the definition of the 'FNS' function and in the program. A point to note is that the names that you give to parameters in the definition of functions are nothing to do with any variables that you might use in the rest of the program. In this sense DEF FNS(A)=A\*A, DEF FNS(Z)=Z\*Z, etc., all define the same function! In a function definition the parameter merely shows what is to happen to the *real* parameter when the function is used – because of this, such a parameter is often called a *dummy parameter*. If you've managed to cope with these ideas, you might like to try to work out what this program's result is:

```

10 DEF FNT(I)=SUM+I
20 SUM=0
30 PRINT FNT(30)
40 SUM=10
50 PRINT FNT(30)

```

In this case the variable 'SUM', used in the expression to define the function, isn't a dummy parameter and it is therefore taken to be a variable in the main program, i.e. 'SUM' in the function definition is the same as 'SUM' in the rest of the program. The idea of a dummy variable gets easier after you have had time to think about it.

To summarise, the rules for forming a user-defined function are:

- (1) Every function that you use must be defined using a DEF FN statement somewhere in the program before it is first used.
- (2) Only numeric user-defined functions are allowed.
- (3) A function definition must include exactly one parameter.
- (4) Any valid BASIC expression can be used in a function definition.

These four rules may seem like a lot to remember but they all accord with common sense. Some examples of functions may help to clarify matters:

```

DEF FNC(X)=(ATN(X)/SQR(1-X*X))+1.5708
DEF FND(N)=INT(RND(1)*N)+1

```

The first function calculates the arcsine function (see the earlier note on the trig functions) and its definition is one of the many to be found in Appendix G of the Oric Manual. The second example will return a random number between 1 and N.

## Subroutines – GOSUB and RETURN

The idea of creating new operations by defining functions is a very powerful one but it is already possible to see its shortcomings. Sometimes it would be an advantage to give a name, not just to one line of BASIC in the form of a user-defined function, but to a whole collection of lines. This is the idea behind a subroutine. A subroutine is nothing more than a group of BASIC statements that can be used as often as required just by writing their 'name'. (In the same way that a single line of BASIC can be used as often as required by using a user-defined function's name.) The trouble with BASIC

subroutines is that they are very limited. You cannot even give a one-letter name to a subroutine. It has to be referred to by the line number of its first line and there is no provision for parameters of any sort. Even with these restrictions the BASIC subroutine is still well worth knowing about and using.

If the lines of BASIC that go to make up the subroutine start at line 'n' then you can use the subroutine by:

GOSUB n

Where GOSUB stands for GO to SUBroutine. Indeed, the action of a GOSUB is very like GOTO in that it transfers control to line 'n'. The difference between a GOTO and a GOSUB is that the GOSUB command causes the Oric to store the line number of the GOSUB in a special area of memory set aside for the purpose. This stored line number is used by the RETURN statement to transfer control to the line following the GOSUB when the subroutine has finished. For example, the effect of a GOSUB and a RETURN on the flow of control is:

```

      10 GOSUB 1000 -----
-->  20 PRINT A
      30 . . .
      40 rest of program
      1000 A=56  <-----
--- 1010 RETURN

```

Line 10 transfers control to the subroutine that starts at 1000, which simply stores 56 in the variable 'A'. The RETURN statement at line 1010 ends the subroutine and automatically transfers control back to line 20.

You can use *any* BASIC statement that you can use elsewhere in a program within a subroutine. In particular, there is nothing to stop a subroutine from using GOSUB and transferring control to another subroutine. If you do this, then the next RETURN will transfer control back to the statement after the most recent GOSUB. In other words, if one subroutine calls another then RETURN behaves as you would expect it to, by transferring control back to the place that each subroutine was initiated, i.e. a RETURN never forgets where it came from!

This pair of instructions GOSUB and RETURN are all that there is to the BASIC subroutine. All the variables in a subroutine are the same as the variables in the rest of the program, there are no

parameters of any kind. As suggested earlier, this might lead you to believe that subroutines are not very useful – this is far from the truth.

## **Using subroutines**

Many programmers believe that the most useful way to use a subroutine is to replace any piece of program that is needed more than once. For example, if in a large program you need to print the same message over and over again, then it is better to turn that line into a subroutine and GOSUB to it every time it is needed. Although this is an important use of subroutines it is often the case that it is a good idea to form parts of a program into subroutines even if each part is only used once! The reason for this is that programs that use subroutines are easier to understand, easier to find mistakes in and easier to modify. This is not the sort of statement that anyone can prove because what is easier in this context is clearly a matter of opinion. You can find an example of this use of subroutines in Chapter 12 of the Oric Manual where a large program is developed. The use of subroutines in writing BASIC programs will be illustrated by the examples in the rest of the book. If you discover some other method of programming that you like better, then no one will be able to argue with you! All I can say is that many programmers agree that subroutines are a good thing!

No matter how you use subroutines there are two applications that you cannot afford to ignore. In Chapter Three the idea of grouping together BASIC statements using the colon was introduced. This was then used in the two extended versions of the IF statement – IF...THEN 'list' and IF...THEN 'list 1' ELSE 'list2' – to avoid complications in the flow of control. The only problem was that a list of BASIC statements on a single line separated by colons soon becomes difficult to read and this limits the usefulness of these forms of the IF statement. Now we have another way of grouping statements together that doesn't become confusing no matter how many statements are involved. So instead of using:

IF 'condition' THEN 'list of BASIC statements'

use

IF 'condition' THEN GOSUB xxx

where xxx is the line number where the 'list of BASIC statements'

used in the first version, starts once it has been converted into a subroutine, i.e. with one statement to a line and ending with a RETURN statement. And instead of using:

```
IF 'condition' THEN *list 1' ELSE 'list 2'
```

use

```
IF 'condition' THEN GOSUB xxx ELSE GOSUB yyy
```

where xxx is the line number where the subroutine containing the BASIC statements of 'list 1' starts and, similarly yyy is the line number where the subroutine containing the BASIC statements of 'list 2' starts. Remember if a list of instructions is more than a few statements long – turn it into a subroutine!

## Chapter Six

# Low Resolution Graphics and Colour

The main delight of programming the Oric is using its graphics and sound! This chapter starts by introducing the sort of graphics that can be produced using `PRINT` and `PLOT` statements. These are usually referred to as *low resolution graphics* and on the Oric are also known as *teletext* graphics and are the same system as used on Ceefax and Oracle. Chapter Seven adds sound and Chapter Eight deals with drawing pictures in finer detail, *high resolution graphics*. You shouldn't be misled into thinking that high resolution graphics are in some way more useful than low resolution graphics. They are not more advanced, just different and it's amazing how often a program works better and is easier to write using low resolution graphics.

Perhaps the biggest challenge in using the Oric is controlling its colour display. In both low and high resolution modes the Oric uses a method called *serial attributes* to determine how something will be displayed on the screen. Serial attributes are a good idea in that they allow a full eight-colour display to be produced using no more memory than a black and white display with the same resolution. However, as the attribute codes that set the colour have to be positioned on the screen, just like any other characters, there are limitations on the way that you can change the display colour. These limitations are not too bad once you understand the logic behind serial attributes and this is described in some detail in this chapter and returned to in Chapter Eight.

### Controlling `PRINT`

So far, we have used the `PRINT` statement to print numbers and strings on the screen, either one to a line or next to each other on the same line. Although this is sufficient for most programs there is

often a need to control exactly where something will be printed on the screen. In Chapter Two the PRINT statement was defined as:

```
PRINT 'print list'
```

where 'print list' was explained to be a list of items each one separated by semicolons. The need for the semicolons is simply to show where one item ends and another begins. For example, PRINT COUNT SUM will try to print a single variable called 'COUNTSUM' (remember blanks are ignored in variable names), while PRINT COUNT;SUM will try to print two variables 'COUNT' and 'SUM' next to each other on a single line.

The Oric actually allows the use of two symbols to separate print items and each one has a different effect on the layout. The semicolon (;) that we have been using since Chapter Two simply means print the next item without leaving any space. Using a comma (,) as a separator instructs the Oric to leave three spaces before printing the next item. For example:

```
PRINT "A","B"
```

will print "A" in column 1 and then leave three spaces so placing the "B" in column 5. Numbers, however, are always printed starting with a single space. For example:

```
PRINT 1;2
```

will print the two PRINT items next to each other but the single space that forms the start of the second number stops the two numbers running together. So rather than 12 being printed as you might expect 1 2 results. In the same way:

```
PRINT A,B
```

will give four spaces between the numbers, the first three are the result of the comma and the fourth is just the space that precedes the printing of all numbers! (As you will recall, A and B are numeric variables, set to 0 by default.)

You can use a print separator more than once in the same statement e.g., PRINT "A",,"B" will cause the Oric to print six spaces between the "A" and the "B". There is one special case that is worth commenting on. If either of the separators is placed at the end of a list of print items, the automatic starting of a new line is suppressed. For example:

```
PRINT "A",
PRINT "B"
```



is the same as PRINT "A","B"  
and:

```
PRINT "A";  
PRINT "B"
```

is the same as PRINT "A";"B"

## TAB and SPC

The use of different 'print list' separators has certainly increased our control over how things are printed on the screen but we still cannot exactly control the space between print items. To achieve this we need something more than different separators. The special function TAB is provided to control exactly the amount of space left between each print item. (It is *special* in the sense that although TAB is written like a function, it produces no value as a result of its use and it can be used only as part of a 'print list'.)

The general form of the TAB function is:

```
TAB('arithmetic expression')
```

and its effect is to move the printing position on by the number of columns given, by the value of 'arithmetic expression'. For example:

```
PRINT "A";TAB(10);"B"
```

will print "A" at column 1 and "B" at column 11. In other words, TAB(10) moves the printing position on 10 columns from the "A". You can have as many TABs in a PRINT statement as you need so:

```
PRINT "A";TAB(10);"B";TAB(20);"C"
```

will print "A" at column 1, "B" at column 11 and "C" at column 31.

Unfortunately the first issue of the Oric's BASIC ROM contained a bug which misplaces the screen position by 12 columns! If you find that a command such as:

```
PRINT TAB(3);"A"
```

fails to work on your Oric then you have an early ROM that contains the bug. Getting around the bug is easy. If you want to move the initial printing position to column X use:

```
TAB(12+X)
```

instead of TAB(X). For the rest of this book the correct form of the

TAB function will be used and it is left to the reader to add 12 to the column number if the Oric that is being used has the bug described above.

Another, and bug-free, way of leaving a number of spaces between print items is the SPC function. SPC(X) will print X spaces on the screen and so for nearly all purposes achieves the same effect as TAB(X). If you want to write programs in Oric BASIC that work on old and new ROM Orics then use the SPC function rather than TAB.

Although TAB and SPC functions are useful it is still difficult to control the printing of numbers and strings down to the exact line and column number. To do this we have to use a completely new command in place of PRINT.

## PLOT

The command:

```
PLOT x,y,'string'
```

will print 'string' on line 'y' starting at column 'x'. So for example:

```
PLOT 10,15,"HELLO"
```

will print HELLO on line 15 with the H positioned in column 10. You can use either a string constant or a string variable to specify the characters that make up 'string'. The only thing to be careful about is that the values of 'x' and 'y' do correspond to a column and line number that is on the screen. The text lines are numbered starting at the top of the screen and the only complication is that the first line is line zero. The columns are also numbered starting from zero with column zero on the far left of the screen (see Fig. 6.1). Thus you should be able to see that the Oric's text screen has column numbers 0 to 38 and line numbers 0 to 26. So for example if you use PLOT 40,10,"HELLO" then you will get an ILLEGAL QUANTITY ERROR because there is no column 40! But:

```
PLOT 3,5,"A"
```

will PLOT "A" on line 5 (i.e. the sixth line down) and column 3 (i.e. the fourth printing position). Normally the Oric prints starting at column 1 but the PLOT command allows you to place characters in column zero. However, column zero has a special role in controlling the colour of the screen so for the moment it is better left unused.

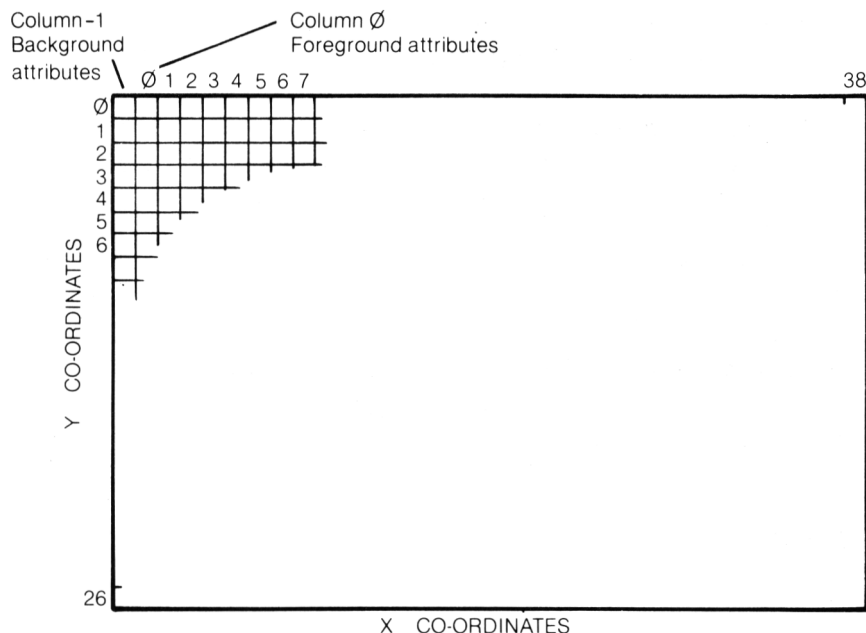


Fig. 6.1. The text screen.

It is worth noticing that PLOT will place a string on the screen without changing anything other than the character locations where the new string appears – this should be compared to the PRINT statement which always clears a line even if it only prints one character at the beginning. For example try:

```
10 PLOT 1,0,"*****"
20 PLOT 5,0,"HELLO"
```

It doesn't matter exactly how many asterisks you use in line 10; they are only there to show the effect of the following PLOT command in line 20. When you RUN this short program you will see HELLO appear within a line of asterisks.

As an exercise in using PLOT try to write a program that draws a square made of asterisks on the screen. In case you have any problems, one of the many possible answers (there is always more than one way of writing a program) is:

```
10 FOR I=0 TO 10
20 PLOT I+10,5,"*"
30 PLOT I+10,15,"*"
40 PLOT I0,I+5,"*"
50 PLOT 20,I+5,"*"
60 NEXT I
```

```

  * * * * *
  *
  *
  *
  *
  *
  *
  *
  *
  * * * * *

```

*Fig. 6.2. Square of asterisks.*

Lines 20 and 30 print the two horizontal rows of asterisks and lines 40 and 50 print the two vertical columns of asterisks.

### **PLOTting numbers**

The command PLOT looks as though it is the ideal replacement for PRINT. Using it you can place strings of characters anywhere on the screen and there seems to be no reason to use the limited PRINT command. However, PLOT has a number of limitations of its own. For example you cannot use more than one string expression per PLOT command. In other words, there is no equivalent of the 'print list' for the PLOT command. If you want to PLOT a number of items then you have to form a single string or single string expression. This is not a serious limitation and apart from not being able to use the print separators or TAB and SPC to introduce additional formatting, a string expression is as easy to use as a 'print list'. A more serious limitation is the restriction to PLOTting strings. Not all of the Oric's variables are strings and to make PLOT a really useful command it is important to find some way of PLOTting the contents of numeric variables. The trouble is that a command like:

```
PLOT 10,15,COUNT
```

won't print the contents of the variable COUNT because PLOT needs a string variable or constant following the last comma (what this command does will be discussed later). The solution is, of course, to convert the contents of the numeric variable to a string using the STR\$ function. Thus the command:

```
PLOT 10,15,STR$(COUNT)
```

will print the contents of the variable COUNT on line 15 starting at column 10. The only problem is that a bug in the first version of the Oric's BASIC ROM caused the STR\$ function to add a single

colour control code as the first character of its result. The effect of this is to change the colour of the number that is printed out. You may not be able to see this colour change on a black and white TV screen but it is very obvious on a colour set. The solution to the problem is to remove the first character from the result of STR\$. That is:

```
10 S$=STR$(COUNT)
20 PLOT 10,15,RIGHT$(S$,LEN(S$)-1)
```

However, if you want to write programs that work on early and later versions of the Oric it is better to check that the first character really is a control character before removing it. The following subroutine will remove any control characters from the front of the string S\$:

```
1000 IF S$="" THEN RETURN
1010 IF ASC(S$)>31 THEN RETURN
1020 S$=RIGHT$(S$,LEN(S$)-1)
1030 GOTO 1000
```

Exactly how this subroutine works will be clear after the description of the Oric's control codes at the end of this chapter. As an example of using subroutine 1000 and the PLOT command consider the problem of printing a list of numbers lined up with a heading. The following program will print the square root of each of the first ten integers:

```
10 CLS
20 PLOT 10,2,"X"
30 PLOT 15,2,"SQUARE ROOT"
40 FOR I=1 TO 10
50 S$=STR$(I)
60 GOSUB 1000
70 PLOT 10,3+I,S$
80 S$=STR$(SQR(I))
90 GOSUB 1000
100 PLOT 18,3+I,S$
110 NEXT I
120 END
```

Remember to make sure subroutine 1000 is also entered before trying to RUN this program! The instruction in line 10 is new – CLS will CLeaR the Screen and set the current printing position to the top left-hand corner of the screen. The rest of the program is straightforward. Lines 20 and 30 print the two parts of the heading.

The FOR loop lines 40 to 110 then prints I and SQR(I) using the PLOT command. Notice the way that the numeric values are converted to string values by lines 50 and 80 and then subroutine 1000 is used to remove control codes (if any) at lines 60 and 90. The PLOT commands in lines 70 and 100 then print the contents of S\$ on the screen in the correct positions. Notice the way the line that the PLOT commands use is incremented by one each time through the loop because of the use of 3+I as the second expression.

### **PLOT and the text cursor**

The PLOT command used in conjunction with STR\$ and the other string functions can be used to exercise complete control over the placing of items on the Oric's screen. However, it is a little more complicated to use than the PRINT command in that you have to convert all of the data values to strings before use and there are no print separators. Perhaps a more serious limitation is that the PLOT command does not move the text cursor or affect it in any way. This means that the screen will not scroll up if you PLOT on the bottom line and it is difficult to control the position on the screen that INPUT data appears when you type on the keyboard. For example, if you use PLOT to display a question in the middle of the screen and then use an INPUT statement to get a value, the question mark printed by the INPUT is unlikely to be at the end of the question! To see this try:

```
10 CLS
20 PLOT 5,14,"HOW OLD ARE YOU"
30 INPUT AGE
40 PRINT AGE
```

The CLS in line 10 clears the screen and sets the text cursor to the top left-hand corner. The PLOT in line 20 displays the question on line 14 but the INPUT statement at line 30 prints its question mark at the current text cursor position – i.e. the top left-hand corner. If you type a number in response to the INPUT it will appear in the top left-hand corner so completing the nonsense that the INPUT statement has made of the attempt at formatting the screen.

There are two solutions to this difficult problem. Firstly, you could abandon the use of INPUT in favour of GET\$. GET\$ will input a single character at a time without printing it on the screen. Using GET\$ in combination with PLOT you can display a question

and get the input without disturbing the screen display. For example:

```
10 CLS
20 PLOT 5,14,"HOW OLD ARE YOU ?"
30 S$=""
40 GET$ A$
50 IF ASC(A$)=13 THEN GOTO 80
60 S$=S$+A$
70 GOTO 40
80 PLOT 23,14,S$
90 AGE=VAL(S$)
100 END
```

The PLOT command in line 20 prints the question but this time complete with question mark. Lines 30 to 90 are the equivalent of the INPUT statement. The GET\$ command in line 40 is used to build up the string in S\$. Once RETURN is detected by line 50, S\$ is PLOTTed at the end of the question by line 80 and the string value converted to a numeric value in AGE using VAL at line 90. Although this simple program will work, it has a number of serious problems as far as the user is concerned. It doesn't allow the input to be corrected because it doesn't recognise the delete key. It doesn't check that only digits have been typed and it doesn't even display what has been typed until RETURN is pressed! All in all, it is a very poor version of an INPUT routine. It is possible to improve the above program to the point that it provides all of the facilities of the INPUT command but this is not easy and the resulting program is very long. However, it does have the advantage of using nothing that could possibly depend on the version of Oric BASIC that it uses – which is more than can be guaranteed of the alternative solution.

A more suitable solution to the PLOT and text cursor problem is to write a subroutine that will move the text cursor to any desired position on the screen. This can be done quite easily but it involves directly altering a number of the Oric's internal memory locations. The only reason that this might cause a problem is that internal memory locations are not guaranteed to remain unaltered in future versions of the Oric's BASIC ROM. However, with this in mind the following subroutine will move the text cursor to line ROW and column COL for both the old and new ROM Orics issued to date. Do not worry about how this subroutine works. DOKE and POKE will be discussed in Chapter Nine.

```

2000 DOKE #12,48039+ROW*40+COL
2010 POKE #268,ROW+1
2020 RETURN

```

Using this subroutine is simple. For example, the previous program to read in the AGE becomes:

```

10 CLS
20 PLOT 5,14,"HOW OLD ARE YOU COL  "
30 COL=23
40 ROW=14
50 PRINT CHR$(17);
60 GOSUB 2000
70 INPUT AGE
80 END

```

Subroutine 2000 is used in line 60 to move the text cursor to the end of the question produced by line 20. The only complication is that the text cursor has to be switched off before it is moved to avoid leaving a non-flashing copy of it at its old location. This is achieved by line 50. Once you have finished with this program you can restore the cursor by pressing CTRL and Q together. Indeed, if you use subroutine 2000 to control the text cursor, there is no need to use the PLOT command at all:

```

10 CLS
20 COL=5
30 ROW=14
40 PRINT CHR$(17);
50 GOSUB 2000
60 INPUT "HOW OLD ARE YOU";AGE
70 END

```

In this case subroutine 2000 is used to move the text cursor to the position that the INPUT statement will print its prompt and then accept the input.

Clearly, it would be possible to use subroutine 2000 together with INPUT and PRINT to control the screen positioning for both input and output without ever using PLOT. However, it is advisable to minimise the use of subroutine 2000 to directly alter the cursor position to make sure that any programs that you write will be as compatible with future version of Oric BASIC as possible.



## Serial attributes

Now that we have a number of ways of controlling where a character or a string of characters will be displayed on the screen it is time to look at the way that the Oric controls how the characters are displayed – *serial attributes*. The idea that lies behind serial attributes is not difficult to understand. The first 32 characters of the Oric's character set are special in that they do not correspond to shapes that can be displayed on the screen. Instead they are interpreted by the Oric as commands that change some aspect (the colour, size, etc.) of the characters that follow on the same line. For example `CHR$(1)` is a command to change the foreground colour to red and any characters that occur on the same line and to the right of `CHR$(1)` will be displayed in red. So, the 32 characters corresponding to `CHR$(0)` to `CHR$(31)` are taken as commands to alter the way that other characters on the same line and to their right will be displayed. These 32 characters are often referred to as *attribute codes*. An attribute code shows as a blank in the current background colour and the space that it occupies cannot be used to print another character without automatically overwriting the code and so removing its effect.

The meaning of each of the attribute codes can be seen in Table 6.1. If you examine this table carefully you will see that the codes fall into four groups:

- codes 0 to 7 set the foreground colour
- codes 8 to 15 set the size and flashing attributes
- codes 16 to 23 set the background colour
- codes 24 to 31 not normally of any use

It makes sense to discuss the attributes that affect colour in a section of their own and similarly the size and flashing attribute codes deserve their own section. However there are a few further general points to be made before moving on to the more specific ones.

## PLOTting, PRINTing and ESC

Using serial attributes is very easy but there are a number of small details concerning the way the `PRINT` statement works that increase their complexity. Suppose that you want to display the two words `HELLO THERE` with the first word in red and the second

*Table 6.1*

Code	ESC	Effect
0	@	Change to black foreground
1	A	Change to red foreground
2	B	Change to green foreground
3	C	Change to yellow foreground
4	D	Change to blue foreground
5	E	Change to magenta foreground
6	F	Change to cyan foreground
7	G	Change to white foreground
8	H	Standard normal height characters
9	I	Graphics normal height characters
10	J	Standard double height characters
11	K	Graphics double height characters
12	L	As 8 but flashing
13	M	As 9 but flashing
14	N	As 10 but flashing
15	O	As 11 but flashing
16	P	Change to black background
17	Q	Change to red background
18	R	Change to green background
19	S	Change to yellow background
20	T	Change to blue background
21	U	Change to magenta background
22	V	Change to cyan background
23	W	Change to white background
24-31		Change display mode – do not use

word in blue. The simplest way of doing this is to first form a string containing the appropriate codes and characters. The attribute code for red is CHR\$(1) and for blue it is CHR\$(4) (see Table 6.1). If you try:

```
10 S$=CHR$(1)+"HELLO THERE"
20 PLOT 4,10,S$
```

You will see both words HELLO THERE displayed in red. The attribute code for red instructs the Oric to display everything to its right using a red foreground colour. To change the colour of THERE to blue it is necessary to embed the blue foreground attribute code just before it. That is:

```
10 S$=CHR$(1)+"HELLO"+CHR$(4)+"THERE"
20 PLOT 4,10,S$
```

Notice that the attribute code for the blue foreground colour i.e. CHR\$(4) now occupies the space between the two words. In other words, the line on the screen is:

column number	4	5	6	7	8	9	10	11	12	13	14	15
display	SP	H	E	L	L	O	SP	T	H	E	R	E
Code	01						04					
colour					red				blue			

where SP indicates that a space in the current background colour is displayed at that location.

The idea of creating a string with the correct attribute codes embedded and then PLOTting it is easy enough to understand but if you try the same program with PLOT replaced by PRINT you will find that it just doesn't work! In other words:

```
10 S$=CHR$(1)+"HELLO"+CHR$(4)+"THERE"
20 PRINT S$
```

just prints HELLO THERE in the usual colours. The trouble is that the PRINT statement interprets characters that correspond to ASCII codes less than 32 in a variety of different ways. For example CHR\$(7) is the attribute code for foreground white but if you try:

```
10 PRINT CHR$(7)
```

you will find that rather than changing the colour of the foreground it makes the Oric go ping!

So that you can include attribute codes within PRINT commands, the Oric recognises CHR\$(27), the code produced by the ESC key, in combination with the standard letters shown in Table 6.1 to stand for the attribute codes. So for example, PRINTing CHR\$(27) followed by "A" will result in the attribute code for red foreground being placed on the screen. You can also enter attribute codes directly by pressing the ESC key followed by the appropriate letter key. To see this, clear the Oric's screen and then move the cursor into approximately the middle using the arrow keys and then type ESC followed immediately by the letter A. Any other characters that you type on the same line will be displayed in red.

Now that the use of the ESC character in PRINT statements has been described, the problem of PRINTing HELLO THERE in two different colours is easy to solve:

```
10 S$=CHR$(27)+"AHELLO"+CHR$(27)+"DTHERE"
20 PRINT S$
```

The CHR\$(27) makes the Oric interpret the next letter as an attribute code as given in Table 6.1. As the first CHR\$(27) is followed by A, the word HELLO is displayed in red and as the second CHR\$(27) is followed by D, the word THERE is displayed in blue. On the early version of the Oric's BASIC ROM this program will not work properly because of a bug that misinterprets a CHR\$(27) printed in column 1. The solution is to avoid column 1 by changing line 20 to:

```
20 PRINT " ";S$
```

which will ensure that the program works on all versions of the Oric. To summarise:

If you are using the PLOT Command, then you can use the attribute codes directly with the CHR\$ function to embed them in a string.

If you are using PRINT commands, you have to use the ESC character, CHR\$(27), followed by the single letter indicated in Table 6.1 instead of the attribute code.

## **Default colour – INK and PAPER**

So far, all of the examples of serial attributes have used the colour codes to illustrate the ideas involved. However, there are a few extra commands and details of operation that are unique to the colour attributes. In particular when you first switch on your Oric what determines that the characters will be displayed black on a white background? The answer is that initially the screen is set up with a column of white background attribute codes, CHR\$(23), and a column of black foreground codes, CHR\$(0), as far over to the left as possible. Indeed, the background codes are stored in a column that cannot be altered by PRINT or by PLOT and is one place to the left of column 0. In this sense it might well be called column -1 (see Fig. 6.1). The codes for the foreground colour are stored in column 0 which cannot be altered by a PRINT statement but can be altered by a PLOT statement. As both columns are as far over to the left as possible, they set the background and foreground colours for the rest of the text on the screen unless another attribute code is present

on the line. You can change these default colours by using the INK and PAPER commands. The command:

PAPER c

will store the background attribute code  $c+16$  in the reserved column -1. Similarly,

INK c

will store the foreground attribute code c in column 0. As both INK and PAPER store the code at the start of each line, the result is that the whole screen changes colour at once. The only exceptions to this are any characters whose colour is controlled by attribute codes on the screen to the right of column -1 or 0. INK and PAPER have no use other than setting the default background and foreground colours. In particular they cannot effect the colour of a single character on the screen.

## The graphics characters - LORES

The PLOT command can obviously be used to place a character anywhere on the screen and, as we saw earlier in this chapter (in the program that prints a square), this can be used to produce limited graphics. This ability only comes into its own when used with the Oric's range of graphics characters. If you look at Table 6.1 you will see that attribute codes 8 and 9 select between the standard character set and a graphics character set. As well as the standard character set of letters, numbers and punctuation, the Oric has a range of graphics shapes all derived from the basic character square which is divided into six parts as shown in Fig. 6.3. Each graphics character has a different combination of foreground and background squares. To see the entire range of characters that the Oric can produce try:

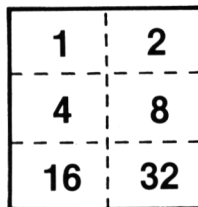


Fig. 6.3.

```

10 FOR I=32 TO 128
20 PRINT " ";CHR$(27);"H";CHR$(I);CHR$(27);"I";
   CHR$(I)
30 NEXT I

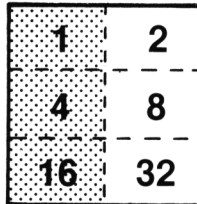
```

Although line 20 looks very complicated, the first CHR\$(27) is followed by H and so this is the attribute code that selects the standard character set and the second CHR\$(27) is followed by I and this selects the graphics characters. Notice that CHR\$(I) can produce either of two possible characters, the one which is displayed being determined by the attribute code to its left.

You could use the graphics characters to make up shapes and lines by looking through the list produced by the previous program and picking the ones that you need. However, there is a simple way of finding the code that will produce any graphics character. If the small squares that make up a graphics character are numbered as shown in Fig. 6.3, then the code of a graphics character can be found by adding together the numbers corresponding to the squares that are to be displayed in the foreground colour, and then adding 32. For example, the vertical bar graphics character shown in Fig. 6.4 has the code 53 and so it can be PRINTed using:

```
10 PRINT " ";CHR$(27);"I";CHR$(53);
```

where, as before, CHR\$(27);"I" is the attribute code that selects the graphics character set and CHR\$(53) is the vertical bar.



$$\text{Code} = (1 + 4 + 16) + 32 = 53$$

*Fig. 6.4.*

You can select between the standard character set and the graphics character set by placing the correct attribute code to the left of the character code. However this method is a little clumsy when in most cases we either want to use mainly the standard character set or mainly the graphics character set. To increase the flexibility of the Oric's display the command:

```
LORES 1
```

will both clear the screen to a black background, set the foreground colour to white and select the graphics character set. The command:

LORES 0

works in the same way as LORES 1 but the standard graphics set is selected. Notice that LORES introduces nothing new in the sense that all it does is to store the appropriate attribute codes on the left-hand side of the screen. Thus, if the screen is made to scroll following LORES 0 or LORES 1 you will see standard text lines filling the screen from the bottom as the attribute codes are scrolled out.

As an exercise in using the graphics characters consider the problem of a better looking square using graphics characters:

```
10 LORES 1
20 FOR I=1 TO 9
30 PLOT I+10,5,CHR$(80)
40 PLOT I+10,15,CHR$(80)
50 PLOT 10,I+5,CHR$(53)
60 PLOT 20,I+5,CHR$(53)
70 NEXT I
```



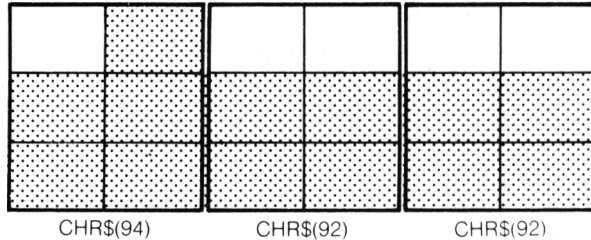
Fig. 6.5. Graphics square.

In this case the graphics character set is selected by the LORES 1 command in line 10. Lines 30 to 60 PLOT either a horizontal bar CHR\$(80) or a vertical bar CHR\$(53) to form the sides of the square. The trouble with this square is that the corners are missing and putting them in is a matter of printing at the correct positions four 'L' shaped graphics characters. This is left for you to remedy.

As you can imagine, drawing more complicated shapes using the graphics characters is very difficult. Fortunately, apart from drawing the occasional 'thick' horizontal or vertical line, the graphics characters are normally only used in small numbers to print special shapes. For example, if you want to print the outline of a ship during a game you could use:

```
PLOT X,Y,CHR$(94)+CHR$(92)+CHR$(92)
```

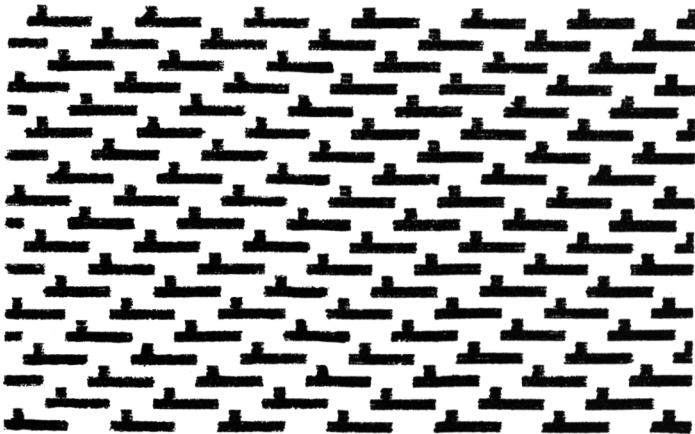
which, assuming that the screen is set up following a LORES 1, will PLOT a ship at column X, line Y.



*Fig. 6.6. Ship graphics.*

If you would like to see a ship move across the screen try:

```
10 LORES 1
20 FOR X=1 TO 34
30 PLOT X,5,CHR$(32)+CHR$(94)+CHR$(92)+CHR$(92)
40 NEXT X
```



*Fig. 6.7. Screenful of ships.*

Notice the space, `CHR$(32)`, left before the first graphics character in line 30. If you want to know what the space is for try leaving it out! If you want to animate a coloured ship then the space can be replaced by a foreground colour attribute. For example to see a red ship change line 30 to:

```
30 PLOT X,5,CHR$(1)+CHR$(94)+CHR$(92)+CHR$(92)
```



## User-defined graphics characters

The range of shapes that can be made up from combinations of the graphics characters, discussed in the previous section, is fairly limited if you want to form small shapes. For example, how would you make up the shape of a man within the space of a single character? Luckily all of the Oric's characters, standard and graphics, can be altered to produce any shape you could want.

Before we can go on to explain how to define new characters, we first have to examine how characters are produced on the screen. Every character that the Oric can display on the screen is in fact produced from a grid of 48 dots arranged into a grid, six dots wide by eight dots high. The pattern of any character depends on which dots in the grid are displayed as foreground and which are displayed as background colour. For example, the letter 'a' is produced by the pattern of dots shown in Fig. 6.8 (f stands for foreground and b for background).

```

b b b b b b
b b b b b b
b f f f b b
b b b b f b
b f f f f b
f b b b f b
b f f f f b
b b b b b b
    
```

Fig. 6.8.

You might find it difficult to see the pattern of the letter 'a' among the 'f's and 'b's but it becomes very clear if each 'f' is replaced by an asterisk and each 'b' is replaced by a blank as in Fig. 6.9.

```

      * * *
      *
    * * * * *
    * * * * *
    *       *
    * * * * *
    
```

Fig. 6.9.

Obviously, if we are going to define the shape that corresponds to a user-defined graphics character, there must be some way of specifying which dots in the six by eight grid are foreground and which are background. The definition of the new character has to be done a row at a time. Each row of dots in the grid is converted into a

number representing the pattern of background and foreground dots. The way that this is done is similar to the way that the code for a particular graphics character was obtained in an earlier section. Each row of the character is composed of six dots and each dot is associated with a number as shown in Fig. 6.10.

32	16	8	4	2	1
----	----	---	---	---	---

*Fig. 6.10.*

The number that corresponds to the pattern of foreground and background dots is obtained by adding together the numbers that are in positions that correspond to foreground dots. So for example the dot pattern `ffbbff` gives  $32+16+2+1$ , or 51, as the number that represents it. After converting all eight rows of dots to numbers, all that remains is to replace the character definition already stored in the Oric's memory. The eight numbers that define the dot pattern for character `x` are stored starting at memory location  $46080+\text{ASC}(x)*8$ , if the character `x` is in the standard character set, and at  $47104+\text{ASC}(x)*8$ , if it is in the graphics character set. (This address is correct for both the 16K and 48K Oric, because the 16K Oric's hardware adjusts addresses that are beyond the end of RAM down to correct values for smaller machine!) Once you know the address that the character definition is stored at, it is easy to replace the current eight numbers that define each row by the new set of eight numbers using the POKE command. The command `POKE a,v` will store the value 'v' in the memory location at address 'a'.

All this will be easier after an example. The little man character mentioned above could be defined using the dot pattern shown in Fig. 6.11.

dot pattern	value
<code>bbffbb</code>	12
<code>bbffbb</code>	12
<code>ffffff</code>	63
<code>bffffb</code>	30
<code>bffffb</code>	30
<code>bfbfbf</code>	18
<code>bfbfbf</code>	18
<code>bfbfbf</code>	18

*Fig. 6.11.*

This definition can replace the existing definition of any of the Oric's characters. So, for example, you could replace the "A" with a man shape by:

```

10 POKE 46080+ASC("A")*8+0,12
20 POKE 46080+ASC("A")*8+1,12
30 POKE 46080+ASC("A")*8+2,63
40 POKE 46080+ASC("A")*8+3,30
50 POKE 46080+ASC("A")*8+4,30
60 POKE 46080+ASC("A")*8+5,18
70 POKE 46080+ASC("A")*8+6,18
80 POKE 46080+ASC("A")*8+7,18
90 PRINT "A"

```

It is possible to shorten this section of program by storing the eight values in a DATA statement but the form given above makes clear the method of redefining each row of dots in turn. It is easy to enter using the Oric's editing facility. Notice that after you have run this program, the A in every line displays as the little man shape. The reason for this is that once you have defined a character the definition holds until you either redefine it or switch the machine off.

User-defined graphics characters are most useful for producing the special shapes that are so essential to any sort of games program. For example, a large dot character could be used as a ball, etc. However, there are some serious uses of user-defined graphics characters, such as showing mathematical or chemical formulae on the screen. In practice useful characters such as the letter A would not be redefined; instead characters that are less often used, such as @, or characters in the graphics set would be redefined. Remember *any* and *all* of the Oric's character set can be redefined.

## Double height characters

If you look back at Table 6.1 you will see that attribute codes 10 and 11 produce double height characters. Unfortunately things are not quite as simple as the existence of a double height attribute code would suggest. What actually happens following a double height attribute code depends on whether the line number is even i.e., 0,2,4..., or odd, 1,3,5... On lines with odd line numbers, only the top half of a character is displayed but enlarged to twice normal size so as to fill the space that a normal character would take. On lines with

even line numbers, only the bottom half of a character is displayed, once again enlarged to twice normal size. You should be able to see that to produce a complete character twice normal size you have to print the same attribute code and characters twice, once on an odd numbered line and once on an even numbered line. For example to print the letter A at twice normal size:

```
1Ø CLS
2Ø PRINT
3Ø PRINT " ";CHR$(27);"JA"
4Ø PRINT " ";CHR$(27);"JA"
```

where the attribute code for double height standard characters is produced by ESC,J. Line 2Ø is required to throw a line of space so that printing starts on line 1, an odd line, and the top half of the letter is printed first.

Although PRINTing everything twice is a workable way of getting double height characters, it is tedious. Fortunately the Oric provides a facility to PRINT everything twice automatically. Following a CTRL D or a PRINT CHR\$(4); everything PRINTed on the screen appears on two lines. To see this, press CTRL and D at the same time, then move the cursor up to roughly the middle of the screen and you will discover that any key you press produces output on two lines. To remove this double printing, simply press CTRL and D at the same time again. The auto double printing facility can now be combined with the double height attribute code to produce:

```
1Ø CLS
2Ø PRINT
3Ø PRINT CHR$(4);
4Ø PRINT " ";CHR$(27);"JA"
```

which will print the letter A in double height without explicitly printing it twice. Notice that even following CHR\$(4) it is important to start printing on an odd line number. You could of course combine lines 2Ø and 3Ø into a single print statement. After running this program, press CTRL and D together to restore normality.

## Flashing characters

All of the Oric's characters, standard, graphics and double height, can be produced as flashing characters. Attribute code 12 produces normal height standard flashing characters, code 13 standard sized

flashing graphics characters, codes 14 and 15 produce double height standard and graphics flashing characters respectively. Using these attribute codes is easy and the only point to notice is that they work in combination with the foreground and background colour codes to produce flashing characters in a range of colours, for example:

```
1Ø CLS
2Ø INK 1
3Ø PAPER 7
4Ø PRINT " ";CHR$(27);"LHELLO"
```

produces a red flashing HELLO.

### Inverse colours

One of the biggest problems in using Oric colour graphics is that if you want a single character to be a different colour from everything else on the screen, then it has to be surrounded by two spaces; a leading space in the form of an attribute code that sets the new foreground colour and a trailing space in the form of an attribute code that resets the foreground colour. If you also want to change the background colour then you would have to add another two spaces. The need to use attributes in this way limits how close two different coloured characters can be on a single line. For example, if you had a green rocket ship and a red rocket ship, the closest that they could approach horizontally without changing colour is one character – a green or a red attribute code would always have to be between them!

There is one way of changing the colour of a shape, however, that doesn't depend on using serial attributes. If  $\text{CHR}\$(x)$  is a character that is being displayed in the current foreground and background colours,  $\text{CHR}\$(x+128)$  will be displayed in the *inverse* foreground and background colours. Unfortunately the PRINT statement will not allow characters with ASCII codes greater than 127 to appear on the screen, so the PLOT command has to be used. For example try:

```
1Ø CLS
2Ø INK 2
3Ø PAPER 4
4Ø PLOT 4,5,CHR$(65)
5Ø PLOT 4,6,CHR$(65+128)
```

will PLOT the letter A (ASCII code 65) in green on blue and then line 50 will PLOT the letter A in inverse colours, which happen to be magenta on yellow. This facility is very useful in that no attribute need be stored to the left of the character to bring about the colour change, but the range of changes is limited. The inverse of each colour can be seen in Table 6.2.

*Table 6.2. Colours and their inverses*

Colour	Inverse
black	white
red	cyan
green	magenta
yellow	blue
blue	yellow
magenta	green
cyan	red
white	black

The main use of inverse colour is to produce large single coloured areas or blocks rather than individual characters. The reason for this is that both the background and the foreground colours are changed and this makes the character location visible as a small solid rectangle as in the above example.

### **Changing a single character location**

It is very often the case that the attribute code stored in a single character location has to be changed to bring about a colour change in a number of characters to its right. To make this easy the Oric provides a special version of the PLOT command:

PLOT X,Y,'attribute code'

which will store 'attribute code' at the screen location given by X and Y. So, for example, PLOT 5,10,17 will store attribute code 17 (a red background) in line 10 at column 5. Notice that the only difference between this form of PLOT and the one that we have been using is that the code doesn't have to be converted to a string. That is PLOT 5,10,17, produces the same result as PLOT 5,10,CHR\$(17) and may

be considered simply as a shorthand for the longer form.

### **Using graphics in games**

We are now in a position to use graphics in programming applications. However a program using what we've learned in this chapter is held over until after we've considered sound, an indispensable adjunct to writing exciting games.

## Chapter Seven

# Sound and Games

The Oric has a bewildering range of commands concerned with sound. As well as the general purpose **PLAY**, **SOUND** and **MUSIC**, it also has a number of pre-programmed sound effect commands such as **EXPLODE**, **SHOOT**, etc. These pre-programmed sounds are attractive and easy to use. However, it can take quite a lot of ingenuity to produce sounds worth listening to using the general commands. The Oric's sound generator is a sophisticated device, with three tone channels and one noise channel and this is reflected in the number and complexity of its **BASIC** sound commands. In the first part of this chapter we will examine some of the ideas involved in using sound to good effect. In the second part, an example of a game involving both sound and graphics will be presented and explained. Although most of this game could have been written at the end of Chapter Six, it is remarkable how much excitement can be added to a game by the careful use of sound.

### **PLAY, SOUND and MUSIC**

One of the confusing things about Oric sound is that the three commands seem to be too many in that they duplicate each other. This is far from true; each command has a very clear purpose and use and once this is understood things become easier. The badly named **PLAY** command doesn't actually produce any sound whatsoever, it merely controls the type of sound the other two commands, **SOUND** and **MUSIC**, will produce. **SOUND** is the fundamental command that will start the Oric's loudspeaker producing tones or noises. **MUSIC** is a slightly more refined command that, while only allowing you to control the tone channels also, lets you specify the pitch in terms of the familiar musical scale. To summarise:



PLAY defines the type of sound produced

SOUND produces a tone or noise of a given pitch

MUSIC produces a tone from the musical scale

This means that the procedure in any sound program is first to use the PLAY command to define the type of sound that you want to produce and then to use either SOUND, if you are producing sound effects or using the noise channel, or MUSIC, if you are playing a tune or don't want to use the noise channel.

The action of the PLAY command is best explained with reference to the MUSIC command because it avoids any complications with the noise channel (which is discussed later). The syntax of the simplest form of the PLAY command is:

PLAY channel,  $\emptyset$ ,  $\emptyset$ ,  $\emptyset$

where 'channel' is a number from 0 to 7 indicating which of the tone channels is 'on' according to Table 7.1.

Table 7.1

Channel	Effect
0	No channel on
1	Channel 1 on
2	Channel 2 on
3	Channels 1 and 2 on
4	Channel 3 on
5	Channels 1 and 3 on
6	Channels 2 and 3 on
7	All three on

(If you know about binary numbers you will notice that 'channel' is simply a three bit number with b0 controlling channel 1, b1 controlling channel 2 and b2 controlling channel 3.) Only channels that have been turned on using PLAY can actually produce any sound although, as we shall see, even channels that are turned off take notice of MUSIC commands. The syntax of the MUSIC command is:

MUSIC channel, octave, note, volume

where 'channel' is a number between 1 and 3, 'octave' is a number

between 0 and 6, 'note' is a number between 1 and 12, and 'volume' is a number between 1 and 15. The MUSIC command will set the specified tone channel to produce a note at a pitch given by 'octave' and 'note' at a volume specified by 'volume'. On the subject of volume, 1 gives the quietest sound, 15 the loudest. The way that 'octave' and 'note' work together to define the pitch of the sound is not difficult to understand if you have even a slight knowledge of music.

The western musical scale is divided into 12 notes each a semitone apart – giving the so called 'chromatic scale'. If you start playing this scale and reach the top note, then instead of going off into the musical wilderness when you play the next higher note, all that happens is that the scale repeats itself but at a higher pitch – or, in other words, in the next octave. The fundamental pitch of the scale that the Oric produces is set by the value of 'octave' and the particular note is selected by 'note'. So, for example, the following program will play the chromatic scale in each of the octaves that the Oric can produce:

```
10 PLAY 1,0,0,0
20 FOR O=0 TO 6
30 FOR N=1 TO 12
40 MUSIC 1,O,N,10
50 WAIT 30
60 NEXT N
70 NEXT O
```

Line 10 turns tone channel 1 on and then lines 20 to 70 play each note at each octave. The only other point worth mentioning is that the WAIT command in line 50 is necessary to set the time that each note lasts for. The command:

WAIT n

will cause the Oric to wait for  $n \times 10$  milliseconds. Rather than sounding a note for a specified time the Oric will continue sounding a note until it is instructed to produce another or until all the tone channels are turned off using PLAY 0,0,0,0.

You may be wondering why you have to specify the tone channel that you want to use in both a PLAY command and a MUSIC command. Surely mentioning the channel in just one command would have been enough? The answer is 'no' if you want to play chords! The MUSIC command sets a tone channel to a given frequency and a given volume even if the channel is off so that you

cannot hear it. If you want to play a chord so that all the notes start together the obvious way to do it is to switch all the channels off, set the pitch and volume of each one and then turn them all on with a single PLAY command. Try, for example:

```
1Ø MUSIC 1,3,11,1Ø
2Ø MUSIC 2,1Ø,3,1Ø
3Ø MUSIC 3,1Ø,7,1Ø
4Ø PLAY 7,Ø,Ø,Ø
```

Lines 1Ø to 3Ø set up the pitch and volume for three notes and then line 4Ø switches all three channels on together, producing a harmonic chord.

The SOUND command works in much the same way as the MUSIC command, apart from the additional feature of being able to control the noise channel as well as the tone channels. The syntax of the SOUND command is:

SOUND channel,pitch,volume

where 'channel' is a number between 1 and 6, 'pitch' is a number between 0 and 65536 and 'volume' is a number between 1 and 15. Once again values of 'channel' between 1 and 3 select one of the tone channels. However, now the pitch of the note is controlled by a single number, 'pitch', which gives a high-pitched note for 0 and lower pitches as the value increases. Although the command will accept a huge range of values for 'pitch', in practice you will find that values between 1 and 300 are the most useful. Above this range all the notes sound the same! Apart from this difference in the way that the pitch is specified you can use the SOUND command in exactly the same way as the MUSIC command. What sets the SOUND command apart from MUSIC is its ability to control the noise channel and this is discussed in a later section.

If you know a little bit about music theory then you can use MUSIC to write your own tunes. However, if you would first of all like to hear what the Oric can do on its own try:

```
1Ø PLAY 1,Ø,Ø,Ø
2Ø MUSIC 1,INT(RND(1)*7),INT(RND(1)*12+1),1Ø
3Ø WAIT 5
4Ø GOTO 2Ø
```

which uses the MUSIC command to produce a random note in a random octave. The noise that this program produces is interesting at first but soon becomes boring. The trouble is that music which is

too random just doesn't sound interesting. Although it is very difficult to introduce enough *order* into the computer-generated music to make it sound anything like traditional music, you can see the overall effect of increasing the order if you try the following program:

```

10 PLAY 1,0,0,0
20 O=4
30 N=6
40 MUSIC 1,O,N,10
50 N=N+SGN(1-RND(1)*2)
60 IF N>12 THEN N=1:O=O+1
70 IF N<1 THEN N=12:O=O-1
80 IF O<0 THEN O=0
90 IF O>7 THEN O=7
100 WAIT 5
110 GOTO 40

```

This plays a infinite sequence of notes that go either up or down by one semitone at most and sounds just a little more like music than the first program. In fact it is not a bad imitation of the *Flight of the Bumble Bee*! Before leaving the subject of random music, it is interesting to hear the effect of changing the note length in each of the above programs. You can do this by changing the value of the WAIT in each program.

Instead of random music you might feel that being able to *play* the Oric is a better idea and indeed it is not difficult to turn the Oric's keyboard into a musical keyboard! Try the following simple program:

```

10 DATA 1,3,5,6,8,10,12,1,3
20 DATA 3,3,3,3,3,3,4,4
30 DIM N(9)
40 DIM O(9)
50 FOR I=1 TO 9
60 READ N(I)
70 NEXT I
80 FOR I=1 TO 9
90 READ O(I)
100 NEXT I
110 PLAY 1,0,0,0
120 GET AS$
130 MUSIC 1,O(VAL(A$)),N(VAL(A$)),10
140 GOTO 120

```

This will allow you to play notes with the top row of number keys from 1 to 9. It works by continually *scanning* the keyboard using GET. Each number is entered as a string variable and is then converted from a string to a number by the VAL function which in some sense is the reverse of the STR\$ function described in Chapter Four. While STR\$(n) will convert the number, n, into a string of digits, VAL(s) will convert a string of digits, s, into a number. The arrays 'N' and 'O' hold the note and octave numbers for the scale. You should now be able to write a program to make every key on the keyboard produce a different note. What is more difficult is to find an arrangement of keys and notes that makes the Oric easy to play!

### Programming tunes

Programming either well-known tunes or even tunes that you have composed is all a matter of working out the sequence of notes and their durations. This is easy if you have the tune written down. If you can find middle C on the music stave then a note drawn on this line corresponds to octave 3 and note 1. Moving up or down by one place on the stave increases or decreases the note number by 2 or 1 (remember to change the octave number if the note number becomes less than 1 or greater than 12) (see Fig. 7.1). The reason for this is that notes sometimes differ by a whole tone and sometimes by only a semitone. The pattern of tone/semitone differences is easy to remember because it is exactly the same as the arrangement of black

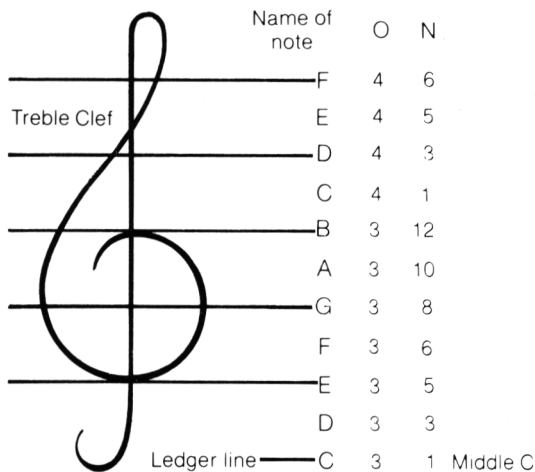


Fig. 7.1.

and white notes on the piano. For example, starting from C gives the following pattern of tone/semitone differences:

C	–	D	–	E	–	F	–	G	–	A	–	B	–	C
	T		T		S		T		T		T		S	

An additional trouble is that most music involves sharps and flats. These are easy to deal with once you realise that a sharp raises the value of the note by one and a flat lowers it by one. For example, C is 1, C sharp is 2 and C flat is 12 in the octave below. The only thing that you have to remember is that if a note is shown as sharp or flat at the start of the music (i.e. in the key signature) then it and all its octaves must be sharpened and flattened.

The well-known beginning of *Hearts of Oak*, apart from being a good tune, could form the basis for a *jingle* suitable for a game involving ships. The first eleven notes can be seen in Fig. 7.2 and converting them to pitch values is easy enough. The three sharp signs

E	A	A	A	A	C <sup>#</sup>	B	A	G <sup>#</sup>	F <sup>#</sup>	E
3	3	3	3	3	4	3	3	3	3	3
5	10	10	10	10	2	12	10	9	7	5

Fig. 7.2. *Hearts of Oak*.

at the beginning apply to all Gs, Fs and Cs in the tune and this rule is best applied by writing the name of each note underneath and then writing a sharp sign by each G, F and C. The pitch values are then assigned, using Fig. 7.1, remembering to add four for a sharp. When converting tunes that have flats in their key signatures you have to subtract one every time a flattened note is played. The resulting pitch values can be seen under the name of each note in Fig. 7.2. Only one thing now keeps us from hearing *Hearts of Oak* and this is the problem of how long each note should be sounded for. Fortunately, musical notation is rigorously logical (after all it was one of the first programming languages)! Time is divided into intervals and a plain ordinary note, like the first in *Hearts of Oak*, should last one interval. The time that a note lasts is shortened by the number of *streamers* drawn on its tail. Each streamer halves the length of the note. For example, the fourth note has two 'streamers', the first of

which shortens it to half an interval and the second reduces it to a quarter. The only complication is that a dot following a note is an instruction to lengthen it by half the time that it would normally last (it makes you wonder how musicians cope!). So the third note would normally be one half a time interval but because it is followed by a dot it has to be sounded for one half plus one quarter i.e. three-quarters. Translating this musical notation into fractions of the time interval, gives the results written under the pitch values in Fig. 7.2. There are two notes that do not occur in *Hearts of Oak* that have to be sounded for twice as long and four times as long. These are included in Fig. 7.3, along with all the other note values.


Note	Time
	4
	2
	1
	$\frac{1}{2}$
	$\frac{1}{4}$
	$\frac{1}{8}$

Fig. 7.3. Lengths of notes.

The time has come to start programming! Each note of the tune now has two numbers associated with its pitch and one with the time that it should sound. This information is best stored in a DATA statement and then read into three variables – one for octave, one for note number and one for duration. Try the following:

```

10 DATA 3,5,1,3,10,2,3,10,.75,3,10,.25,3,10,1,4,2,.75
20 DATA 3,12,.25,3,10,1,3,9,.75,3,7,.25,3,5,1.5,999,999,999
30 TEMPO=50
40 PLAY 1,0,0,0
50 READ O,P,T
60 IF P=999 THEN STOP
70 MUSIC 1,O,P,10
80 WAIT T*TEMPO
90 GOTO 50

```

The DATA statement is terminated by three values of 999 and this is used to detect the end of the tune. The variable 'TEMPO' sets the length of the fundamental time interval, in this example half a second, but you might like to experiment with other values.

## Resting

You may think that the notes of the tune *Hearts of Oak* played by the previous program are a little run together. The solution to this problem is to insert a pause between each note. This can be done by switching the sound off using the PLAY  $\emptyset, \emptyset, \emptyset, \emptyset$  command in between each note and then WAITing a short time. For example, add:

```
75 PLAY 1, $\emptyset, \emptyset, \emptyset$ 
85 PLAY  $\emptyset, \emptyset, \emptyset, \emptyset$ 
86 WAIT 1
```

to the previous program and the improvement is immediate! You can use the PLAY  $\emptyset, \emptyset, \emptyset, \emptyset$  command together with WAIT to leave pauses or *rests* in any sequence of notes.

## Pre-programmed sound effects

The Oric provides four different pre-programmed sound effect commands – EXPLODE, PING, SHOOT, and ZAP. Producing these sounds in a program is simplicity itself – just use the appropriate command and follow it by a WAIT that is sufficiently long to allow the sound to come to an end naturally. If you want to carry on doing some computing while the sound effect is being produced then you can omit the WAIT as long as you don't try to produce another sound until the current one has ended. The biggest problem with the pre-programmed sounds is that you cannot alter their volume and they tend to be very loud. Although each of the sounds has a name that suggests its application, e.g. ZAP is the sort of noise that a laser gun makes, it is worth considering them for other applications. For example the EXPLODE sound make a very good imitation of something falling in water when added to appropriate graphics!



## The sound of noise

The Oric's sound generator has a single noise channel that can be played on its own or mixed with any of the three tone channels. To use the noise channel we first have to turn it on using a slightly extended form of the PLAY command:

PLAY tone channel,noise channel,0,0

The 'tone channel' part of the command selects a combination of tone channels as before. The new part of the command is 'noise channel' which is a number between 0 and 7 with the same meaning as 'tone channel' except that it selects the combination of channels that the noise source will be mixed with. This idea of mixing the noise channel with the tone channels is the most difficult Oric sound feature to understand. If the noise channel is mixed with a particular tone channel then any sound command which refers to that channel will also produce the noise source and, as we shall see in a moment, any reference to the noise channel will also produce the tone channel. You can deduce from this that to produce a pure tone on a channel you must not have selected it as a channel to be mixed with the noise channel. Likewise if you want to produce pure noise you must have switched off the tone channel that it is being mixed with. Before we look at some examples, it is necessary to extend the SOUND command so that it can set the 'pitch' of the noise channel. This is quite easy as the syntax of the SOUND command stays the same, but the 'channel' number used previously increases its range to 1 to 6. Channel numbers from 1 to 3 set the pitch of the corresponding tone channel but numbers from 4 to 6 set the pitch of the noise channel. For example, if you have used a PLAY command that mixes the noise channel with tone channel 1 then SOUND 4,20,4 will set the noise channel to a pitch of 20 and allow you to hear the noise along with the tone on channel 1 if it is switched on. However SOUND 1,20,4 will set the tone channel to a pitch of 20 and allow you to hear the noise channel and the newly set tone channel if it is switched on. The only problem is that the idea of a noise channel having a 'pitch' is an odd one! In fact the noise channel can produce a 'shhhhhhhing' noise at any of 32 pitches. For example, to hear the noise channel on its own at each of its pitches all we have to do is mix it with tone channel 1 and then turn this tone channel off:

```

10 PLAY 0,1,0,0
20 FOR I=0 TO 31
30 SOUND 4,I,10
40 WAIT 5
50 NEXT I
60 GOTO 20

```

Line 10 turns tone channel 1 off and mixes the noise channel with it. Lines 20 to 50 change the pitch of the noise channel and allow you to hear the whooshing effect produced. To see, or rather hear, the effect of not turning the tone channel off change line 10 to `PLAY 1,1,0,0`. Now you will hear the same noise channel changing pitch and making the whoosing but you will also hear a tone of constant pitch – this is the tone on channel 1. You can independently change the pitch on both channels without any trouble. To hear this add:

```

35 SOUND 1,31-I,10

```

to the program along with the change to the play command and, if you can stand the head-spinning noise that results from the noise channel rising in pitch while the tone channel falls, you are obviously cut out to be a researcher into Oric sound!

The noise channel is the obvious source of most sound effects. For example, the whooshing noise in the last program would make a good rocket take off sound and the `EXPLODE` sound is clearly based on the noise channel. The two main ways of making the new effects are by manipulating the noise pitch and the noise volume. For example, try:

```

10 PLAY 0,1,0,0
20 SOUND 4,2,8
30 WAIT 10
40 SOUND 4,20,12
50 WAIT 5
60 GOTO 20

```

which produces a noise like a helicopter by sounding noises of different pitch and volume. However, if you really want to play with sound then you cannot ignore the last feature that the Oric offers – the envelope.

## PLAYing an envelope

The PLAY command has one last surprise in store for us. The last two parameters can be used to select one of seven predefined volume envelopes and durations. This gives the full and final form of the PLAY command as:

PLAY tone channel,noise channel,envolope,envolope duration

A volume envelope is not a difficult idea, it is simply a graph of volume with time but it can be difficult to know what any given envelope sounds like. Fortunately the Oric reduces the choice to a manageable 7 (see Fig. 7.4) selected by the value of 'envelope' in the PLAY command. The first two are 'finite length' envelopes, that is

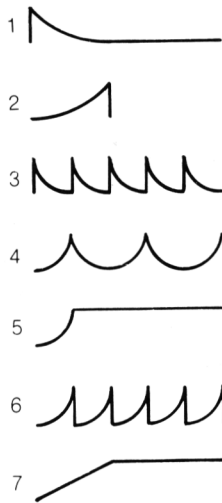


Fig. 7.4. Envelopes.

they do not change the volume of the note in a periodic manner but have a definite start and a definite end. The first one rises sharply and then declines slowly, the second rises slowly and then declines quickly. To make a sound whose volume is controlled by either of these envelopes, all you have to do is use a PLAY command that specifies the envelope and turns on the tone or noise channel in question. Then any SOUND or MUSIC command which specifies a volume of zero will use the specified envelope. For example, to hear the effect of envelope 1 try:

```
10 PLAY 1,0,1,1000
20 SOUND 1,50,0
```

```
30 WAIT 30
40 GOTO 10
```

Notice that to make the envelope repeat it is necessary to carry out the **PLAY** command each time. The last parameter in the **PLAY** command alters the duration of the envelope. In other words, it alters how long it takes the note to die away for envelope 1 or how long it takes the note to build up to full volume in envelope 2. There is no choice but to experiment with this parameter to get the effect that you want. Long times give echoes or warbles and short times give percussive or rough sounds.

The final five envelopes are periodic, that is they repeat the same overall envelope shape until a **PLAY 0,0,0,0** command silences the channel in question. The effect of the envelope duration is much more difficult to gauge for these envelopes in that it alters the repeat rate. Try:

```
10 PLAY 1,0,4,100
20 SOUND 1,80,0
```

which give a fast repeat and then replace line 10 by

```
PLAY 1,0,4,500
```

which gives a slower rising and falling of volume. It is important to keep in mind that an envelope only alters the volume of a note – the pitch is always determined by the **SOUND** or **MUSIC** command. You can use envelopes with the noise channel just as easily. To hear an example try:

```
10 PLAY 0,1,4,100
20 SOUND 4,20,0
```

which produces a familiar sound using only the noise channel.

## Pitfalls

The Oric's sound commands are very logical and as long as you keep a clear head they are easy enough to use. There are some common mistakes, though, that are worth looking out for. For example, the keyboard bleep will interfere with any sounds that you have gone to a lot of trouble to produce, so it is always a good idea to turn it off with a **CTRL** and **F**. Another common problem is getting the envelope duration so short that it introduces a low frequency buzz rather than changing the volume.

There is no easy way to predict what sort of noise a sound program will produce when you run it, but after some practice and experimentation you will get better at guessing!

### Attack the saucer - the SCRN function

The game listed below uses most of the BASIC commands that have been introduced in this chapter and Chapter Six.

```

10 LORES 0
20 GOSUB 10000
30 FOR X=0 TO 38
40 GOSUB 20000
50 GOSUB 30000
60 PLOT X,25,CHR$(3)+"@"
70 NEXT X
80 FOR X=38 TO 0 STEP -1
90 GOSUB 20000
100 GOSUB 30000
110 PLOT X,25,CHR$(3)+"@ "
120 NEXT X
130 GOTO 30

1000 T=15
1010 F=0
1020 FX=10
1030 FY=10
1040 POKE 46080+ASC("@")*8+0,12
1050 POKE 46080+ASC("@")*8+1,12
1060 POKE 46080+ASC("@")*8+2,12
1070 POKE 46080+ASC("@")*8+3,12
1080 POKE 46080+ASC("@")*8+4,30
1090 POKE 46080+ASC("@")*8+5,63
1100 POKE 46080+ASC("@")*8+6,0
1110 POKE 46080+ASC("@")*8+7,0
1120 FOR I=0 TO 7
1130 POKE 46080+ASC("$")*8+I,INT(RND(1)*64)
1140 NEXT I
1150 RETURN

2000 F$=KEY$
2010 IF F$="" THEN RETURN
2020 PLOT FX,FY," "
```

```

2030 FX=X
2040 FY=25
2050 F=1
2060 RETURN

3000 T=T+SGN(RND(1)*2-1)
3010 IF T<5 THEN T=T+1
3020 IF T>25 THEN T=T-1
3030 PLOT T,5,CHR$(1)+"*** "
3040 IF F=0 THEN WAIT 5:RETURN
3050 PLOT FX,FY," "
3060 FY=FY-1
3070 PLOT FX-1,FY,CHR$(7)+"^"
3080 PLAY 1,0,0,0:SOUND 1,FY*4,10
3090 IF FY<4 THEN F=0:PLOT FX,FY," ":PLAY 0,0,0,0:
    RETURN
3100 IF SCRN(FX,FY-1)<>ASC("***") THEN RETURN
3110 GOSUB 4000
3120 F=0
3130 RETURN

4000 EXPLODE
4010 FOR I=1 TO 10
4020 PLOT FX-1,FY,CHR$(INT(RND(1)*8))+ "$"
4030 PLOT FX,FY," "
4040 NEXT I
4050 RETURN

```

The game itself is relatively straightforward to play. An alien flying saucer, in the form of three asterisks, moves rather jerkily across the screen. A ship, whose purpose is to attack the alien, moves rapidly backwards and forwards at the bottom of the screen. A missile can be launched at any time from the attacking ship by pressing any key. Once a missile has been fired it moves up the screen accompanied by a whistling noise, increasing in pitch, until it either misses the saucer or hits it with a resulting explosion. If at any time during the flight of a missile another key is pressed, then the first missile is erased from the screen and a new missile fired at the saucer.

The program has been written as a small collection of subroutines and is not particularly difficult to understand. It is easier to follow the main part of the program after a description of each subroutine. Subroutine 1000 sets up the user-defined graphics character for the attacking ship (lines 1040-1110) and the explosion (lines 1120-1140).

The attack ship is defined using the method described in Chapter Six but the explosion is defined as a random pattern of dots using a FOR loop. Subroutine 1000 also initialises some of the variables used in the rest of the program.

Subroutine 2000 checks to see if any key has been pressed and fires the missile. If no key has been pressed then control is returned to the main part of the program (line 2010). If any key has been pressed then any existing missile is removed from the screen by PLOTting a blank (line 2020) at the current missile position stored in 'FX' and 'FY'. Then the current missile position is set to the current position of the attacking ship (lines 2030-2040) and variable 'F' is set to 1 to indicate that a missile has been fired and is in flight.

Subroutine 3000 moves the saucer a random amount to the right or left, prints the missile if one is in flight and checks to see if it has hit the saucer. Lines 3000-3030 are responsible for moving the saucer. Notice the checks to stop it from moving off the edge of the screen in lines 3010 and 3020. The printing of the saucer in line 3030 also serves to remove the old saucer from the screen because of the attribute code at the left-hand end of the string of asterisks and the space at the right-hand end. Lines 3040-3080 look after moving the missile. Line 3040 checks to see if there is a missile in flight (i.e.. F=1). If there isn't, control is passed back to the main part of the program. The WAIT instruction is included to make the attack ship move at the same rate even if there isn't a missile in flight. Lines 3060-3070 move the missile up the screen by one line. Line 3060 blanks out the old missile and line 3070 prints it at its new position. Line 3080 makes a sound that increases in pitch as the missile moves higher up the screen. Lines 3090-3100 test to see if the missile has hit or missed the saucer. Line 3090 checks to see if the missile's position is such that it has passed the saucer and is about to go off the screen. If this is the case, a blank is printed to remove the missile and the variable 'F' is set to zero to indicate that there are no missiles in flight. Line 3100 uses the SCRN function to discover which character is at the next screen location that the missile will move into. It is used in line 3100 to discover if the character just above the missile is an asterisk. The function:

SCRN(X,Y)

returns the ASCII code of the character at screen location X,Y. If it is, then the missile is about to hit the saucer and the explosion subroutine 4000 is called.

Subroutine 4000 will produce an explosion at X,Y by printing an

explosion character in random colours while making a sound using EXPLODE.

Now that all the subroutines have been described, the working of the main part of the program is easy to understand. First, subroutine 1000 is called to initialise everything. The main work of the program is done by the two FOR loops 30–70 and 80–120. The first FOR loop moves the attack ship to the right one place at a time. Each time the attack ship moves, subroutine 2000 is called to check for a ‘fire missile’ command and subroutine 3000 is called to move the saucer and the missile. The second FOR loop moves the attack ship to the left but otherwise it is identical to the first FOR loop. Notice the way that each PLOT command includes a foreground attribute code to set the colour of the shape. Using this simple method the saucer is red, the attack ship is yellow and the missile is white.

This concludes the description of this short games program. If you study it to the point that you are sure that you understand it, the way to find out if you’re right is to try to modify it! The game would be made much more exciting by the addition of only a few very simple features. You could, for example, add a routine to keep a score of the number of saucers hit, or give the attacking ship only a limited number of moves before the saucer fires a missile back at it! Try experimenting with these suggestions and your own ideas. After all, the only way to learn to program is to program!



## Chapter Eight

# High Resolution Graphics

The Oric's *high resolution graphics* makes it possible to draw with a resolution of 240 points horizontally by 200 points vertically. This makes it easy to draw fine lines and circles using a range of new 'high resolution' commands. However, techniques such as user-defined graphics characters are not something to be forgotten in favour of this apparently more powerful graphics mode. User defined graphics are very often the best way of approaching a project and the Oric allows us to have the best of both worlds by allowing mixing of high resolution and low resolution techniques.

The Oric's high resolution screen uses the same method of controlling colour, that is serial attributes, as the text screen and this reduces the colour resolution to 40 horizontal 'blocks' by 200 lines. If you understand the way that low resolution colour works you should have no trouble with high resolution colour.

### The high resolution screen

The Oric uses a very similar method to produce high and low resolution graphics. However the high resolution screen is distinct from the text or low resolution screen. To change to the high resolution screen all that is necessary is to use the command:

**HIRES**

This instructs the Oric to use the extra memory set aside for the high resolution screen. If you enter **HIRES** in direct mode you will discover that not all of the high resolution screen is high resolution – there are three lines text at the bottom. This is very useful because you can use these lines to give messages during high resolution programs.

High resolution graphics differ from low resolution graphics in

that it is possible in high resolution graphics to use commands that will change the colour of a single point on the screen. Text or low resolution commands only allow you to change all of the dots within a character location i.e. a rectangle 6 dots wide by 8 dots high. If you are going to use commands that change a single point then there must be some way of defining which point. In the same way that a character location was picked out in low resolution graphics by giving its column and line number you can pick out a single point in high resolution graphics by stating which column and row of dots it is in. The columns of dots are numbered from zero starting at the far left, and so the column number, or *x co-ordinate* as it is called, ranges from 0 to 239. (Recall that there are 240 dots horizontally.) The rows of dots are numbered from zero starting at the top of the screen and so the row number, or *y co-ordinate* as it is called, ranges from 0 to 199. (There are 200 dots vertically.) Any dot on the screen can be specified by giving two numbers, its *x co-ordinate* and its *y co-ordinate*. It is usual to write these two numbers as a pair with the *x co-ordinate* first. So 0,0 specifies the dot in the top left-hand corner and 239,199 is the bottom right-hand corner. After a little practice using *x,y co-ordinates* will become second nature.

The only other thing to know about the high resolution screen at this stage is how to get rid of it! If you want to return to the text screen then simply use the command:

TEXT

Similarly LORES 0 or LORES 1 will return you to the text screen initialised for low resolution graphics with the standard character set or the graphics characters set respectively.

## **CURSET and the graphics cursor**

The high resolution graphics commands are easier to understand when they are working with only two colours and so a consideration of high resolution graphics in colour is left to a later section. Just as with low resolution graphics there are two sorts of dots on the Oric's high resolution screen – foreground dots that show in the current foreground colour and background dots that show in the current background colour.

When you first change to high resolution graphics using the HIRES command the Oric sets the foreground colour to white and the background colour to black!

The simplest high resolution command is:

`CURSET X,Y,FB`

This changes the dot at the x co-ordinate given by X and the y co-ordinate given by Y to either a foreground dot or a background dot depending on the value of FB. The action of FB is:

value of FB	
$\emptyset$	Change dot to background
1	Change dot to foreground
2	Invert dot
3	Do nothing

The first two values of FB are easy enough to understand:

`CURSET X,Y, $\emptyset$`

changes the dot at X,Y, to a background dot and:

`CURSET X,Y,1`

changes the dot at X,Y to a foreground dot. However the action of:

`CURSET X,Y,2`

depends on the existing dot at X,Y. If the dot is initially a foreground dot then it will be changed to a background dot. If it is initially a background dot then it will be changed to a foreground dot. In this sense a value of 3 for FB will cause `CURSET` to *flip* or *invert* the type of dot at X,Y. This action is useful if you want to change a point in such a way that the change will make it show on the screen even if you don't know its current state. For example, you won't see the effect of setting a point to a foreground point if it is already a foreground point! The final value of FB and the action it produces is easy to understand but it is difficult to see the need for it. The command:

`CURSET X,Y,3`

does not alter the dot at X,Y in any way whatsoever! What it does, however, is to move the *graphics cursor*. The graphics cursor works in a similar way to the familiar text cursor but it cannot be seen. It generally can be said to mark the position of the last high resolution dot to have been referred to in a high resolution command – although there are one or two exceptions to this. So the command:

`CURSET X,Y,3`

does have an effect – it moves the invisible graphics cursor to the dot at X,Y without changing anything on the screen. (Why you might want to do this will become clear after the other high resolution commands have been introduced.) The other versions of the CURSET command, corresponding to values of FB equal to 0,1 and 2, also move the graphics cursor to the point X,Y.

Try the following program to investigate and familiarise yourself with the way that CURSET works in conjunction with the possible values of FB:

```
1Ø HIRES
2Ø INPUT X,Y,FB
3Ø CURSET X,Y,FB
4Ø GOTO 2Ø
```

If you enter a value for X and Y that takes the graphics cursor outside the screen you will get an ILLEGAL QUANTITY ERROR. For an automatic demonstration try:

```
1Ø HIRES
2Ø X=INT(RND(1)*24Ø)
3Ø Y=INT(RND(1)*2ØØ)
4Ø CURSET X,Y,1
5Ø GOTO 2Ø
```

which draws dots at random.

## **DRAW and CURMOV**

The trouble with CURSET is that it only changes a single point at a time and while it is possible to draw lines and shapes using nothing but CURSET it would be tedious. To make a high resolution graphics easier to use Oric BASIC includes:

**DRAW X,Y,FB**

which produces a straight line. The starting position of the line is the current position of the graphics cursor and its end is X dots to the right and Y dots down. As in the case of CURSET, the value of FB determines what happens to the dots that lie on the line. That is a value of FB equal to Ø will set all the dots on the line to background points, a value of 1 will set them to foreground points, a value of 2 will cause the dots to be inverted and 3 will leave the dots unchanged. Following a DRAW command the graphics cursor is left at the end

point of the line. For example, if the last CURSET was CURSET 0,0,3 which moves the graphics cursor to 0,0 the command:

```
DRAW 100,100,1
```

will produce a line of foreground dots from the point 0,0 to 100,100. But if the last CURSET was CURSET 50,50,3 the line would start at 50,50 and end at 150,150. It is important to notice that DRAW uses co-ordinates in a way that is completely different from CURSET. The command CURSET X,Y,FB moves the graphics cursor to the point X,Y and then plots a point. The command DRAW X,Y,FB moves the graphics cursor X units horizontally and Y units vertically and then draws a line between the old position of the cursor and the new. The way that a DRAW uses co-ordinates is usually referred to as *relative co-ordinates* because they are relative to the current position of the graphics cursor. The clearest indication that DRAW X,Y,FB is different from CURSET X,Y,FB is in a command such as:

```
DRAW -10,10,1
```

which leaves the graphics cursor 10 units to the left and 10 units down. Negative co-ordinates are not allowed in CURSET!

As an example of the way that relative co-ordinates can make a problem easier to solve, consider trying to write a subroutine that will draw a rectangle W dots wide and H dots high with its top left-hand corner at X,Y. One possible solution is:

```
3000 CURSET X,Y,1
3010 DRAW W,0,1
3020 DRAW 0,H,1
3030 DRAW -W,0,1
3040 DRAW 0,-H,1
3050 RETURN
```

If you would like to see this subroutine in use then add it to the following main program which uses it to draw random rectangles:

```
10 HIRES
20 X=INT(RND(1)*230)
30 Y=INT(RND(1)*190)
40 W=INT(RND(1)*(220-X))+1
50 H=INT(RND(1)*(180-Y))+1
60 GOSUB 3000
70 GOTO 20
```

The way that the DRAW command uses relative co-ordinates is

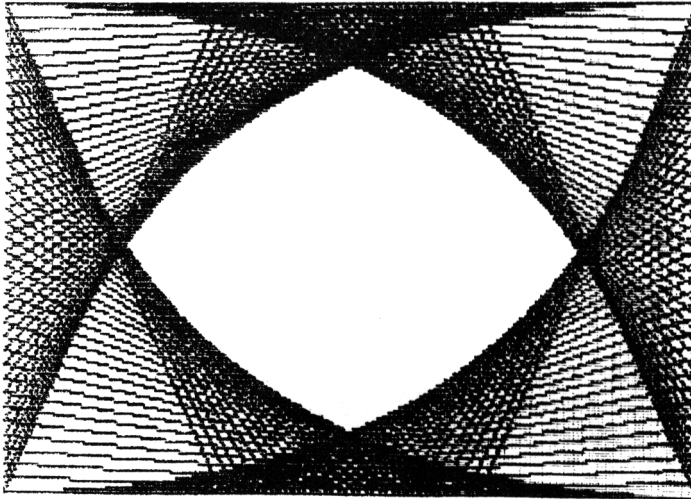
very useful for drawing a number of connected lines but it is often very difficult to DRAW a line between two given points. However, the following combination will draw a line between the point X1,Y1 and X2,Y2:

```
CURSET X1,Y1,FB:DRAW X2-X1,Y2-Y1,FB
```

To see the sort of thing that DRAW can do, try the following program:

```
10 HIRES
20 FOR I=1 TO 199 STEP 4
30 CURSET 0,I,1
40 DRAW 239-I,-I,1
50 CURSET I,0,1
60 DRAW 239-I,I,1
70 CURSET 0,199-I,1
80 DRAW 239-I,I,1
90 CURSET I,199,1
100 DRAW 239-I,-I,1
110 NEXT I
```

The output from this program is reproduced in Fig. 8.1. You will find that you get different effects by altering the values of the STEP in line 20.



*Fig. 8.1. String pattern.*

Relative co-ordinates are so useful that Oric BASIC has another form of the CURSET command that will accept relative co-ordinates. The command:

**CURMOV X,Y,FB**

has the same effect as CURSET but the cursor is moved X dots to the right and Y dots down.

## CIRCLE

The command CIRCLE R,FB will draw a circle centred on the current position of the graphics cursor and radius R. Once again the value of FB determines what will happen to the dots on the circle in the usual way. For example:

**CURSET 100,50,3:CIRCLE 40,1**

draws a circle centered at 100,50 and radius 40. As an example of the circle command, the following program draws random circles, as illustrated in Fig. 8.2:

```

10 HIRES
20 R=INT(RND(1)*50)+1
30 X=R+INT(RND(1)*(239-2*R))
40 Y=R+INT(RND(1)*(199-2*R))
50 CURSET X,Y,3
60 CIRCLE R,1
70 GOTO 20

```

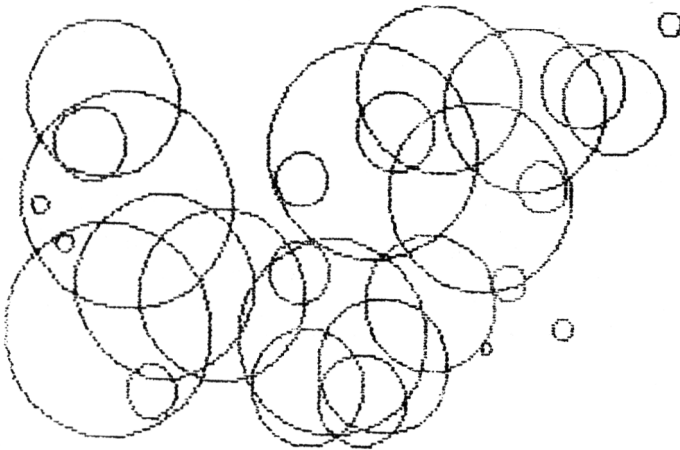


Fig. 8.2 Random circles.

Notice that when drawing random circles you have to be careful not to go off the edge of the screen. This is allowed for in this program by the inclusion of  $-2*R$  in lines 30 and 40.

### **Dotted lines - PATTERN**

Normally when drawing lines or circles the Oric will draw solid lines. However, you can change to dotted lines of your own design using the PATTERN X command. The exact pattern of dots produced depends on the value of X. In practice the most useful forms of the command are:

PATTERN 85	Small dots
PATTERN 51	Larger dots
PATTERN 31	Even larger dots!

Other values of X can be used to produce lines with uneven dots and dashes but you can try these out and experiment to find something that suits your application.

### **Using colour - FILL, INK and PAPER**

Colour in high resolution is determined using the same serial attribute method familiar from low resolution graphics. However, in low resolution graphics the attribute codes were stored in a single character location and affected all of the characters to their right and on the same line. In high resolution graphics the attributes still affect all of the dots to their right and in the same row but where and how are the attribute codes stored? The answer lies in the fact that each row of dots on the high resolution screen is divided into groups of six. These groups are called *character cells* and they are the high resolution equivalent of a character location. Each character cell uses one memory location to store the pattern of dots that it displays. There is nothing stopping us from storing an attribute code in the memory location that normally stores the dot pattern for a character cell. If you do store such a code then the effect is similar to the low resolution case. The dots that are normally controlled by the memory location show as background dots and all the dots to the right of them are affected by the attribute. The only trouble is that all of the high resolution commands that we have looked at so far control single points with no reference to character cells. Storing a



value in the memory location that controls the six dots that correspond to any given character cell relies on the use of a new command, FILL.

The command FILL 1,1,c will store the attribute code 'c' in the memory location corresponding to the character cell that the graphics cursor is currently in. For example:

```
CURSET 0,0,3:FILL 1,1,17
```

will store the attribute code 17 (red background) in the memory location corresponding to the character cell in the top left-hand corner of the screen. The reason for this is that the CURSET command first moves the cursor to the first dot in the character cell concerned, but it could just as well have been the any of the six. So:

```
CURSET X,0,3:FILL 1,1,17
```

with X any of 0 to 5 will produce the same effect and result in attribute code 17 being stored in the character cell in the top left-hand corner. A more striking demonstration can be seen by running:

```
10 HIRES
20 FOR X=0 TO 199 STEP 1
30 CURSET X,X,3
40 FILL 1,1,17
50 NEXT X
```

which moves the graphics cursor one row down and one dot to the right each time through the FOR loop and uses the FILL command to set a red background. You will notice that the red moves horizontally across the screen in jumps of six dots. This illustrates the fact that a whole character cell has to be used to store an attribute code and so colours can only change on character cell boundaries.

You can store any of the attribute codes described in Chapter Six in any of the character cells using CURSET and FILL. However, it is not easy to use full eight colour high resolution graphics. For example, suppose you want to plot a single dot in yellow irrespective of the colour of any other dot on the screen. To make sure it was yellow you would have to use FILL to place a yellow foreground attribute code in the character cell immediately to its left. If the dot in question is at X,Y then the character cell to its immediate left (if there is one) is at X-6,Y. So the following program will plot yellow dots:

```

1Ø HIRES
2Ø INPUT X,Y
3Ø CURSET X-6,Y,3
4Ø FILL 1,1,3
5Ø CURSET X,Y,1
6Ø GOTO 2Ø

```

If you try to plot a dot at 50,10 and then 56,10 you will immediately see the major difficulty in using Oric colour in this simple way. The dot at 50,10 will appear yellow because of the yellow foreground attribute placed to its left by the FILL at line 4Ø. However when the dot at 56,10 is plotted, the dot at 50,10 disappears because the new yellow foreground attribute for 56,10 is stored in the same character cell as the dot 50,10 and this makes all of the dots in this cell show as background points – including 50,10. The solution, of course, is to not place the new yellow attribute code for the second point on the screen. After all if 50,10 is yellow there must be a yellow attribute code to its left and this will also make sure that 56,10 is also yellow. This, however, depends on there not being another foreground attribute stored in a character cell between the two points and in general this is not something you can be sure of! Another feature of colour in high resolution graphics can be seen if you plot the same two points but in the reverse order – that is 56,10 first followed by 50,10. In this case you will discover that 50,10 doesn't appear on the screen. The reason for this is that Oric's graphics commands will avoid altering any character cell that contains an attribute code in an effort not to upset any colour scheme already present. All in all it is very difficult to use high resolution graphics in a mixture of colours without paying attention to what colour everything already on the screen is. There are one or two cases where the nature of the display makes colour easier to handle. For example if you restrict yourself to one colour per row then you can easily use all eight colours on the screen at the same time. Also if you keep high resolution objects at least 6 dots apart then you are free to use the gaps between them to store attribute codes. In practice the restrictions are not as bad as you might anticipate and you may be surprised at how often a display organisation facilitates colour handling. Where this is not the case our advice is to stick to two colours!

As an example of a particular high resolution colour display try the following program which produces a striped display:

```

1Ø HIRES
2Ø Y=Ø

```

```

30 FOR C=16 TO 23
40 FOR I=1 TO 25
50 CURSET 0,Y,3
60 FILL 1,1,C
70 Y=Y+1
80 NEXT I
90 NEXT C

```

This program uses FILL to place each of the background colour attributes into the far left of groups of 25 rows so producing horizontal stripes.

You may be wondering why the FILL command is:

```
FILL 1,1,c
```

The answer is that the general form of the FILL command is:

```
FILL R,N,C
```

which will store the attribute code 'C' in N consecutive character cells on each of R rows – the first character cell is, as before, indicated by the current position of the graphics cursor. Thus:

```
CURSET 0,0,3:FILL 1,1,1
```

will store a single attribute code for foreground red in the top left-hand corner.

```
CURSET 0,0,3:FILL 10,1,1
```

will store the same attribute code at the start of each of the first ten rows on the screen.

Another way of setting a number of attribute codes at the same time is to use INK and PAPER. In high resolution mode the commands INK c and PAPER c will store the appropriate attribute codes at the far left-hand side of the screen. This is all very simple but how do you change the colour of the three text lines at the bottom of the screen? The answer is that any INK and PAPER commands that you issue while in TEXT or LORES modes will remain in effect after a HIRES command. So if you want the three text lines to merge into the default high resolution screen try:

```

10 TEXT
20 INK 7
30 PAPER 0
40 HIRES

```

which produces a white on black text screen and then changes to the HIRES screen which by default is also white on black.

### **Inverse colours**

In low resolution graphics storing a character code plus 128 at a character location will cause the Oric to display the character but with inverse colours used for the foreground and background dots. This is also true of the high resolution screen. If you use FILL to store a code of 128 in a character cell then all dots are set to background dots and these show in the inverse of the current background colour. Any subsequent foreground dots that are plotted within the character cell also show in the inverse of the current foreground colour. As this inverse colour is not obtained by storing attribute codes to the left of the dots affected it is one way of changing colours without leaving a character cell with all background points on the screen.

### **Characters in high resolution - CHAR**

The three lines of text at the bottom of the high resolution screen are often all that is required to print prompts, etc., during programs that use high resolution graphics. However it is often necessary to place text actually on the high resolution screen for example to label graphs. The command:

```
CHAR A,S,FB
```

will draw the character whose ASCII code is in A, using the shape defined in the character set given by S. The value of S can either be 0, for the standard character set, or 1 for the graphics character set. The value of FB determines what happens to the foreground dots in the characters shape in the usual way. The character is drawn so that its top left-hand corner is at the current position of the graphics cursor.

To plot a text string on the high resolution graphics screen requires the repeated use of the CHAR command. For example:

```
10 HIRES
20 AS="I AM A HIGH RESOLUTION ORIC"
30 CURSET 10,20,3
```

```

40 GOSUB 4000
50 CURSET 10,28,3
60 DRAW 162,0,1
70 END

4000 FOR I=1 TO LEN(A$)
4010 CHAR ASC(MID$(A$,I,1)),0,1
4020 CURMOV 6,0,3
4030 NEXT I
4040 RETURN

```

Subroutine 4000 will plot the contents of the string A\$ on the screen starting from the current position of the graphics cursor.

As you can imagine, being able to plot character shapes on the high resolution screen extends itself to user-defined characters. In fact one of the main uses of the CHAR command is to plot small user-defined shapes on the high resolution screen. However there is one small detail of operation to be careful about. When a high resolution screen is being used the memory locations that store the character definitions are different from when a text screen is in use. The easiest way around this problem is to always define any user-defined characters *before* selecting a high resolution display. If you do this everything will work because the Oric automatically moves all the character definitions to the new location whenever you use a HIRES command.

## Using and not using high resolution – GRAB and RELEASE

If you are not going to use the high resolution screen then you can make the memory that it normally occupies available for BASIC programs by using the GRAB command. Similarly, if you want to re-allocate the memory to the high resolution screen you can use the RELEASE command. However you must be careful when you use RELEASE that the high resolution screen doesn't overwrite part of your program.

## Finding out what's on the screen – POINT

In the same way that SCRNB could be used to discover what character is displayed on the screen at any character location, the function POINT can be used to find out what sort of dot is on the

screen at any given high resolution position. The general form of the POINT function is:

POINT(X,Y)

Where X is the x co-ordinate and Y is the y co-ordinate of the position being examined. The value returned by POINT is -1 if there is a foreground dot at the specified co-ordinates and 0 if there is a background dot. The sort of thing that POINT is used for is similar to the use of SCRIN in the saucer program given in Chapter Seven.

### Using high resolution graphics

Although there are only a few high resolution graphics commands it can be difficult to see how they can be used to produce effective displays. As always, the best way to learn is to have a look at some examples and then try to write your own programs.

The first example uses the GET function to control the position of a dot on the screen. By pressing the appropriate arrow keys you can draw shapes on the screen:

```

1Ø HIRES
2Ø X=1ØØ
3Ø Y=1ØØ
4Ø GET A$
5Ø IF ASC(A$)=8 THEN X=X-1
6Ø IF ASC(A$)=9 THEN X=X+1
7Ø IF ASC(A$)=1Ø THEN Y=Y+1
8Ø IF ASC(A$)=11 THEN Y=Y-1
9Ø CURSET X,Y,1
1ØØ GOTO 4Ø

```

You can see a sample of the output of this program in Fig. 8.3. You could try to add some improvements of your own such as diagonal movements and being able to move from one place to another without drawing a line.

The second example is based on a set of patterns discovered by the nineteenth century French physicist, Lissajous and aptly called 'Lissajous figures':

```

1Ø HIRES
2Ø T=Ø
3Ø T=T+Ø.1

```

```

40 X=50*(1+SIN(1.1*T))
50 Y=50*(1+COS(T))
60 CURSET X+50,Y+50,1
70 GOTO 30

```

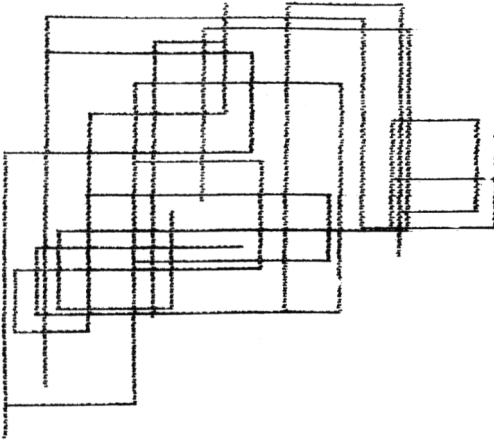


Fig. 8.3. Etch-a-sketch.

The output of this program can be seen in Fig. 8.4. You can produce a range of different patterns by changing the value 1.1 in line 40.

The next example is a program that plots the graph of  $\text{SIN}(x)/x$ . This produces a particularly interesting shape, as you can see in Fig. 8.5. One difficulty to be aware of is that  $\text{SIN}(x)/x$  is impossible to work out when  $x$  is zero so this point has to be carefully left out of the graph.

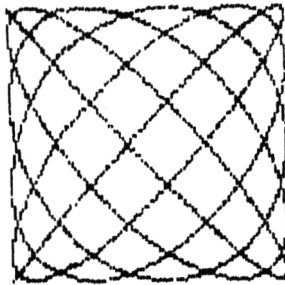


Fig. 8.4. Lissajous figure.

```

10 HIRES
20 FOR I=1 TO 230
30 IF I=115 THEN GOTO 70

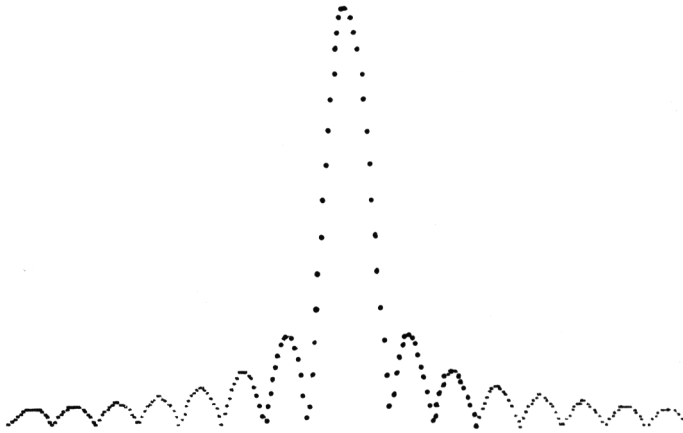
```

```

40 X=(I-115)/5
50 Y=150-150*SIN(X)/X
60 CURSET I,Y,1
70 NEXT I

```

The final example in this chapter is a program that will plot the graph of any equation. The idea is very simple, just work out the values of the equation over the range specified and plot points corresponding to each X and Y values. The only trouble is how do



*Fig. 8.5.* Plot of  $\text{SIN}(X)/X$ .

you make sure that the X and Y values of the function fall into the permitted 0 to 239 and 0 to 199 that will keep the graph on the screen? The answer is that both the X and Y values have to be changed, they have to be 'scaled' to fit the screen. The scaling for the X values is easy because we know beforehand both the maximum and minimum values. If the maximum X value is XBIG and the minimum X value is XSMALL then to scale the X values to fall in the range 0 to 239 we use:

$$XS = \frac{239 \cdot (X - XSMALL)}{(XBIG - XSMALL)}$$

The rationale behind this equation is not difficult to understand. If you subtract XSMALL from X it ranges between 0 and XBIG-XSMALL. If you then divide by (XBIG-XSMALL) the result ranges between 0 and 1. Finally multiplying by 239 gives a quantity that ranges between 0 and 239 which is what is actually required. The same reasoning can be used to scale the Y values but we don't know the values of YBIG and YSMALL. The only way that



these values can be found is to first calculate the equation for each value of X and see what the largest and smallest values of Y are. The only other question is how many values of X is the equation worth calculating for? The answer is that each value of X should differ by an amount that moves the plotted point on by one screen dot. So the X step size should be:

$$S=(XBIG-XSMALL)/239$$

After this very long discussion, it is now time for the program:

```

10 HIRES
20 GOSUB 1000
30 GOSUB 2000
40 GOSUB 3000
50 END

1000 DEF FNA(X)=X*X
1010 INPUT "Draw graph starting at X=";XSMALL
1020 INPUT "Last X value to be graphed =";XBIG
1030 RETURN

2000 X=XSMALL
2010 YSMALL=FNA(X)
2020 YBIG=YSMALL
2030 S=(XBIG-XSMALL)/239
2040 FOR X=XSMALL TO XBIG STEP S
2050 Y=FNA(X)
2060 IF Y>YBIG THEN YBIG=Y
2070 IF Y<YSMALL THEN YSMALL=Y
2080 NEXT X
2090 RETURN

3000 FOR X=XSMALL TO XBIG STEP S
3010 Y=FNA(X)
3020 CURSET (X-XSMALL)/(XBIG-XSMALL)*239,
      199-(Y-YSMALL)/(YBIG-YSMALL)*199,1
3030 NEXT X
3040 RETURN

```

The program consists of three subroutines. Subroutine 1000 defines the equation that the user wants to see the graph of and the range of X values that should be covered. Subroutine 2000 finds the values of YBIG and YSMALL. Finally subroutine 3000 actually plots the graph of the equation. The only unexplained part of the

program is the subroutine 20000 that finds YBIG and YSMALL. This is not difficult to understand. At the start YBIG and YSMALL are both set to the same value, the value of the equation when X equals XSMALL. The equation is then worked out for each value of X that will be plotted and the value of Y is compared with the current values of YBIG and YSMALL. Obviously if Y is larger than the current value of YBIG it should be used to replace the current value. In other words YBIG is the largest value of Y that 'we have seen so far' and after we have seen all the values, it becomes the largest of all values. The same reasoning holds for YSMALL but it is only changed if Y is smaller than the current value. In other words YSMALL is the smallest value we have seen so far.

When using this program it is important to keep a few things in mind. Firstly, you can type in any equation involving X, such as  $X^2$  or  $\sin X$ , in the function definition at line 10000 but the more complicated the equation, the longer the program will take to work it out. Secondly, you can specify any values for XBIG and XSMALL and the graph will still fit on the screen but you might not see a very interesting curve. To demonstrate the program, you might like to try the following equations by typing them in as function definitions at line 10000:

$X^2$	XBIG=10	XSMALL=-10
$X^2X-25X$	XBIG=8	XSMALL=-8
$\sin(X)$	XBIG=10	XSMALL=0
$\sin(X)+\sin(2X)$	XBIG=10	XSMALL=0

When using graphics in your own programs, the best strategy is actually to limit your use of high resolution graphics to those occasions when they perform an essential function. In other words, it is advisable to start off using low resolution graphics but to keep an eye open for situations where high resolution graphics can actually do the job better. The most common way of using the Oric is in low resolution with user-defined graphics but often the mixture of high resolution and user-defined graphics (via the CHAR command) is very successful.

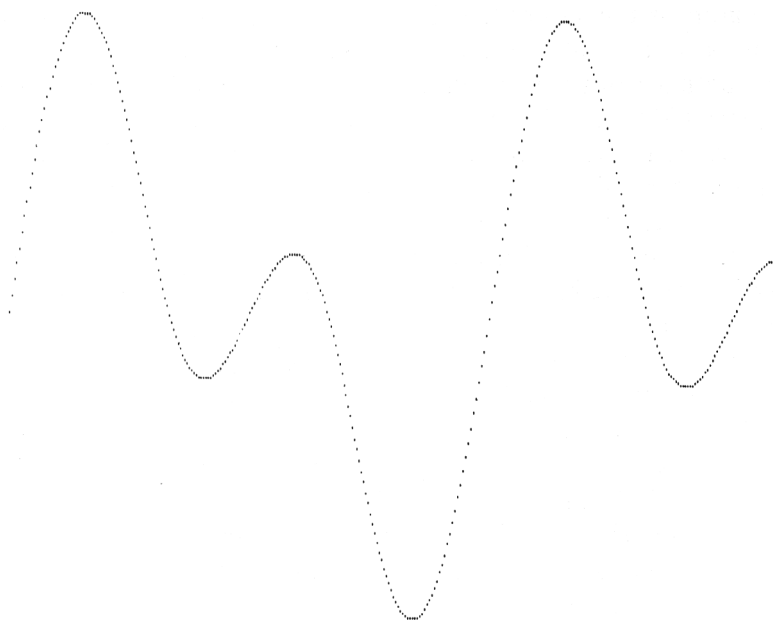


Fig. 8.6. Graph of  $\sin(X) + \sin(2X)$ .

## Chapter Nine

# Logic and Other Topics

In this chapter a number of different topics are introduced. It would be misleading to think of this as a collection of advanced topics just because they have been left until near the end. Using the BASIC and other information dealt with in earlier chapters, you should by now find it possible to write any program that you want to. However, there are facilities on the Oric that, although not entirely necessary, do make things easier or go beyond what can be done from simple BASIC. This chapter collects together these extras and explains a little of how they work.

The first topic to be covered has the rather daunting title of *Boolean logic* and introduces the commands AND, OR, and NOT. The second area deals with commands such as PEEK and POKE which affect the inner working of the Oric.

### Logic and the conditional expressions

In everyday speech we often say things like, 'did you buy apples and oranges?', 'do you prefer tea or coffee?'. The use of words like *and* and *or* are so common that we rarely stop to think about them. It would obviously be a great advantage if the use of *and* and *or* could be extended to BASIC conditional expressions. Luckily, most versions of BASIC do allow the use of these everyday concepts and in Oric BASIC you can write expressions such as:

```
A<0 AND B=3  
A<0 OR B=3
```

The meaning of each of these expressions is in line with the usual English meaning of *and* and *or*. The first of the above expressions is true if both of the conditions 'A<0' 'B=2' are true and the second expression is true if either of the two conditions is true. As well as

AND and OR the Oric also allows the use of NOT, which simply changes the value of a conditional expression from true to false and vice versa. For example, '3=2' is false but 'NOT 3=2' is true.

You can combine AND, OR and NOT with any of the conditions we met in Chapter Four to make more complicated expressions that evaluate to one of the values *true* or *false*. (As you already know, from Chapter Three, *true* is represented by -1 and *false* by 0 in Oric BASIC which allows you to use them very easily in arithmetic expressions.) Conditional expressions that include AND, OR or NOT are usually called logical expressions and we can now re-write the definition of the IF statement as:

IF 'logical expression' THEN 'BASIC statements'

or

IF 'logical expression' THEN 'BASIC statements' ELSE 'BASIC statements'

For example, if you want to check that you're not about to use an x co-ordinate that goes outside the screen area, you could use:

IF X<0 OR X>239 THEN ...

in place of the two IF statements that would be required without the use of OR.

Forming logical expressions to test for *overall* conditions is usually straightforward. However, there are a few traps that even experts fall into. If you want to translate the English statement - 'A equals B and C' then you must repeat the condition =. In other words, you must write:

A=B AND A=C

and not use

A=B AND C

which will give a result that depends on whether C is 0 or -1. You should also be careful when using NOT. For example:

NOT(A=B AND A=C)

isn't the same as

NOT(A=B) AND NOT (A=C)

To see that this is the case try working the two expressions out for a

few values of 'A', 'B' and 'C'. The moral is that you should always beware of using logical expressions without thinking about *exactly* what they mean.

To close the subject of logical expressions it is worth introducing the idea of a *truth table*. If you consider the logical expression:

### A AND B

where 'A' and 'B' are variables that are either 0, for false, or -1 for true. You can draw up a table that lists the result of the expression for all possible values of 'A' and 'B' as follows:

A	B	A AND B
0	0	0
0	-1	0
-1	0	0
-1	-1	-1

Such a table is called a truth table because it lists the conditions under which the logical expression is true or false. You can draw up truth tables for any logical expression and this is one way to check that you understand what is happening. For example, OR and NOT have the following truth tables:

A	B	A OR B	NOT A
0	0	0	-1
0	-1	-1	-1
-1	0	-1	0
-1	-1	-1	0

The OR that we have used so far is not entirely equivalent to the English word 'or'. Most uses of the English 'or' mean 'one or the other but not both' For example, 'You can have jam or marmalade' means that you can pick one but not (normally) both! However, the logical OR means that you can have either one or both (look at the truth table if you are unsure of this). The logical OR is more properly called the *inclusive OR* because it includes the possibility of both. The usual English 'or' is known as the *exclusive or* because it excludes the possibility of both. You can make up a logical expression that is equivalent to the exclusive or:

$$\text{exclusive or} = (\text{NOT}(\text{A}) \text{ AND } \text{B}) \text{ OR } (\text{A} \text{ AND } \text{NOT}(\text{B}))$$

as can be seen from the truth table:

A	B	(NOT(A) AND B) OR (A AND NOT(B))
0	0	0
0	-1	-1
-1	0	-1
-1	-1	0

To make programs that use logic easier to read the Oric provides two predefined variables: TRUE, which is automatically set to -1 and FALSE, which is automatically set to 0. Using these two variables you can write statements such as:

IF 'condition' =TRUE THEN GOSUB 5000

where 'condition' is a variable that is set to 0 or -1 somewhere else in the program. It is also worth mentioning that the ORIC treats any non-zero value as true.

### Inside the Oric – PEEK, POKE, DEEK, DOKE, CALL and USR

It may come as something of a surprise to learn that BASIC includes a number of commands that allow access to the inner workings of the Oric. The reason why this might seem strange is that all the BASIC that we have looked at so far has done its best to avoid getting involved with details of how the machine carries out commands. However, there are some applications where the normal instructions of BASIC are in some way deficient. For example, they might be too slow or fail to take account of some important feature of the machine. To allow the programmer to find a way around such difficulties, most versions of BASIC include instructions that allow you to gain access to the inside of the machine.

Although, from the point of view of the BASIC programmer, the Oric seems to do its arithmetic in terms of decimal numbers, in fact it works things out in a more fundamental system called *binary*. Humans count in decimal simply because they have ten fingers. If we had only two fingers then we would count in the same way that computers do, in binary. Although it is not important for a BASIC programmer to know very much about binary, it is important if you want to use the facilities of the Oric directly. A binary number is simply a number that contains only zeros and ones. For example 01001 is a binary number. We have already met binary numbers in action when converting foreground background dot patterns to decimal numbers. If instead of writing f and b, standing for foreground and background, you write 1 and 0 then the pattern of

ones and zeros that results is a binary number. Working out the decimal number that represents the pattern of dots is nothing more than converting the binary number to decimal! Explaining the theory and practice of binary numbers is beyond the scope of this book and would take us into areas well away from BASIC. However, the subject is not as difficult as you might believe and there is never any reason to worry about the arithmetic involved as this is one area that your Oric is always ready to help with! Once you have come to terms with binary numbers you will find it useful to use yet another sort of number – hexadecimal. Although the Oric will not accept binary numbers directly it will accept and print hexadecimal numbers. A hexadecimal number is signified by preceding it by #. For example #FF is hexadecimal for 255 in decimal. You can also use the HEX\$ function to convert decimal numbers to a hexadecimal in string form.

We learned very early on that a variable is a named area of computer memory. However, it is sometimes necessary to side-step this method of using computer memory and use in preference direct access, via PEEK and POKE. PEEK is a function that will return the contents of a memory location and POKE is a command that will alter the contents of a memory location. It is as simple as that, except that you need to know how to specify which memory location and what sort of number can be stored in a memory location. The first problem is easily solved because the Oric, like all computers, numbers all its memory locations sequentially starting from zero. So PEEK(543) will return the contents of the five hundred and forty-third memory location. The second problem is also easily solved once you know that a memory location can store a binary number with up to eight zeros or ones in it and, as 11111111 evaluates to 255 in decimal, this is the largest number that can be held in a single memory location. So POKE 1000,200 will store 200 in memory location 1000. However, POKE 1000,600 will give an error message because 600 is greater than 255. In general, to use PEEK and POKE you have to have a knowledge of what is stored where inside your Oric and this is often not easy to find out. For this reason, PEEKing and POKEing are best avoided unless you are absolutely sure that you know what you are doing. We have already used POKE in a number of applications, for example to change the shape definitions of characters in Chapter Six. As an example of using PEEK consider the related problem of discovering what the dot pattern of any particular character currently is. To do this all we have to know is where the eight memory locations that store the eight rows of dots



are in memory. However, this problem has already been solved to enable us to POKE new dot definitions. The address of the Ith row of dots of the character in C\$ is:

$$\text{address} = 46080 + \text{ASC}(\text{C\$}) * 8 + \text{I}$$

if the character is in the standard character set. A program to examine and print the contents of each of the eight memory locations is now easy to write:

```

10 INPUT "WHICH CHARACTER";C$
20 FOR I=0 TO 7
30 ADDR=46080+ASC(C$)*8+I
40 DAT=PEEK(ADDR)
50 PRINT DAT
60 NEXT I
70 PRINT
80 GOTO 10

```

The range of numbers 0 to 255 that can be stored in a single memory location is very restrictive. This restriction is overcome by using more than one memory location to store a number. For example two memory locations can be used to store number in the range 0 to 65535. This range is large enough for most purposes and the Oric provides modified forms of PEEK and POKE to enable data to be retrieved and stored using two memory locations as easily as one. The command:

DOKE address,value

will use the two memory locations at 'address' and 'address+1' to store 'value' and:

DEEK(address)

will examine the two memory locations at 'address' and 'address+1' and return the value stored in them.

The commands USR and CALL are very special BASIC commands in that they transfer control out of BASIC and into a *machine code* program stored somewhere inside the Oric. Machine code is a completely new, and vast, topic and until you want to get involved in it USR and CALL will be of little interest to you. However, if you do feel like getting involved in yet another computer language they are your route out of BASIC!

**Controlling BASIC – HIMEM, CLEAR, POP and PULL**

There are a number of Oric BASIC commands that are concerned with altering the way that BASIC works. For example we have already met GRAB and RELEASE which allow BASIC to use or give back the memory normally set aside for the high resolution graphics screen. Similarly the command:

HIMEM address

will stop BASIC using memory above 'address'. The reason why you might want to reserve memory in this way is to store machine code programs.

The command CLEAR will remove any data stored in variables; that is it sets variables to zero and strings to the null string. It also removes any arrays that you might have dimensioned and so gives you the opportunity to redimension them.

The commands POP and PULL allow you to use BASIC in a way that was never intended! The best way to explain POP is by an example. Suppose that you follow the suggestion given earlier in this book and write all your programs as collections of subroutines. Then a problem that sometimes arises is that an error or condition occurs in a subroutine that makes it either impossible or pointless to continue with the rest of the program. However, rather than just printing an error message and stopping, it might be possible to return control to the main program and proceed to another part of the program. This is easy if the subroutine where the condition occurred was called by the main program. In this case a simple RETURN does the trick, but what if the subroutine was used by another subroutine that was called by the main program. In this case a RETURN will only transfer control back to the previous subroutine and not to the main program. The command POP will solve this problem by removing all trace of the last GOSUB that occurred. Following POP a RETURN transfers control back to the point in the program that you would normally get to by carrying out two RETURNS in a row. In other words, POP:RETURN transfers control not to the subroutine that called the current subroutine, but to the subroutine (or main program) that called the subroutine that called the current subroutine! You may think this description has the potential to confuse – it does and this is why the POP command is not something to use very often. It is very difficult to follow the workings of a program that includes POP commands and this in turn can make it difficult to debug and difficult to alter. However,

there are occasions when the POP command is the easiest and even clearest way of working your way back up a collection of subroutine calls without having to *pass through* each one in turn. As an example of POP try the following:

```

10 GOSUB 1000
20 PRINT "BACK IN MAIN PROG"
30 END

1000 PRINT "IN 1000"
1010 GOSUB 2000
1020 PRINT "RETURNED TO 1000"
1030 RETURN

2000 PRINT "IN 2000"
2010 RETURN

```

If you run this program then you will see the expected sequence of subroutine transfers:

Main → 1000 → 2000 → 1000 → Main

However, if you add:

```
2005 POP
```

to the program the sequence of transfers is:

Main → 1000 → 2000 → Main

In other words, the RETURN at line 2010 transfers control right back to the main program without going through subroutine 1000.

The command PULL performs the same service as POP but for the point that a REPEAT UNTIL loop returns to. When the Oric encounters an UNTIL statement it either passes control back to the most recent REPEAT statement or it ends the loop by passing control to the next statement. Following a PULL command an UNTIL statement will either end the loop or transfer control back to the REPEAT statement before the one that it would normally return to. For example:

```

10 REPEAT
20 PRINT "LOOP ONE"
30 REPEAT
40 PRINT "LOOP TWO"
50 UNTIL I=0
60 UNTIL I=0

```

This program consists of two simple loops that never come to an end (because 1 will never equal 0!). When you run the program you will see “LOOP ONE” printed once and then “LOOP TWO” over and over again indicating that the first UNTIL at line 50 transfers control to line 30 as you would expect. However, if you now insert:

45 PULL

you will find that the two messages alternate and this indicates that the UNTIL at line 50 is now transferring control to the first REPEAT statement at line 10. As you can see, PULL is potentially even more confusing than POP and should be avoided if at all possible.

The ideas introduced in this chapter are difficult to absorb in the abstract. As with all the other BASIC commands discussed in this book, the only real way to come to terms with them is to use them in your own programs. At present you may not be able to imagine occasions when you need to rely on Boolean logic or commands such as PEEK and POKE. They do have their uses, however, and once you discover that you actually need one of these rather esoteric commands you will both understand how it works and recognise its value.

## Chapter Ten

# Towards Better Programming

The programming examples in this book have all been presented as complete and working programs. However, if you have tried writing your own programs then you will know that for a program to work first time is a very rare event! In the first part of this chapter the subject of making programs work or *debugging* is considered. When you first start to learn to program there is enough to think about in just solving a problem and producing a program that works without worrying too much about the quality of the program. As you become more familiar with BASIC and with programming in general, however, you will find that as well as being able to tackle more difficult problems you will be able to examine your programs more critically. The second half of this chapter discusses how you can make your programs more reliable.

### Finding bugs

Whenever you write a program there comes the moment when you have enough of it complete to type RUN and expect it to do something useful. Usually something goes wrong and it doesn't perform as you would expect it to. The next question is how to go about correcting what you have written to make it work – in other words, how do you find the bug in a program? Sometimes the problem is easily solved because the Oric gives an error message that names a particular line and listing that line reveals an obvious mistake, such as a mistyping. However, on many occasions the error message names a line that is perfectly correct Oric BASIC. In this case the problem lies in a line that defines something that the line mentioned in the error message uses. For example, if you forget to dimension an array the error message will mention the first line that uses it.

Program bugs that cause error messages to spring up are generally easy to find by just looking at the program. The trouble really starts when a program runs without causing an error message but just doesn't behave as you would expect it to. Even in this case just looking at the program is sometimes sufficient to find the bug, but if you have been staring at the program for more than a few minutes the chances of finding the problem in this way are very small. Because by inspecting a program you can find simple bugs, many programmers think that this is the best method of finding more difficult bugs! This is far from the truth. The only way to find a difficult bug is to *do* something, just staring at the program will only give you a headache. The most important thing to realise is that debugging is an activity. You must try to gather as much information about the way that the program *is* working and compare this with the way that you *expect* the program to be working. There are two things that you need to check to discover where a program is going wrong – that the values stored in the variables are what you would expect them to be and that the order in which the statements are carried out is correct. If both of these things agree with what you would expect then the program is correct and if it still doesn't do what you want it to you are using the wrong method!

Oric BASIC provides a number of debugging aids that will help you discover if the program is behaving as you would expect. Firstly you can check that the flow of control is correct by using the trace function. Following TRON (TRaceON) the Oric will print the number of each line that it obeys and in this way you can follow the execution of a program. The only trouble is that if you enter a loop following TRON you will be inundated with screenfuls of line numbers! The correct way to use the trace commands is to embed them in the program that you are debugging. Place TRON commands at the points in the program where you want to check the flow of control and TROFF (TRaceOFF) commands before any loops, or anywhere else that the quantity of line numbers generated would be more than you could cope with.

A more controlled method of following the flow of control through a program is to use the STOP command. When the Oric encounters a STOP command it stops executing the program and prints the current line number. This information can be used to follow the flow of control through a program. In addition, while the program is stopped you still have access to all of the variables that the program has used. For example, you can find out what is stored

in any variable by entering a PRINT statement in direct mode. To see this in action try:

```
10 INPUT A
20 IF A>0 THEN GOTO 100
30 STOP
40 PRINT A
50 END
100 STOP
110 PRINT A
```

If you type in a positive number you will see the message

“BREAK IN 100”

displayed on the screen. You can then find the value of A by entering

PRINT A

You can even change the value of A by assigning to it in direct mode. After investigating the values stored in the variables the program can be restarted by typing:

GOTO x

where ‘x’ is the line number of the statement that follows the STOP. In the case of the above example, this is GOTO 110. By placing STOP commands carefully and investigating the values stored in variables, it is possible to find out exactly what a program is doing and where it isn’t doing what it should there lies a bug!

## **Good programming style**

There are many different ways to write a program and as you gain in knowledge and experience you should try to write the best program that you can. The only question is ‘what is a good program?’ Obviously a good program must do what you intend it to – that is, it must work – but after that what makes a program ‘good’ is a matter of opinion. In the early days of computing the two most important characteristics of a program were how fast it worked and how much memory it took. However, as the hardware of computing becomes cheaper and more powerful, program performance in terms of speed and size becomes less important. This said, it has to be admitted that there will always be applications that need special attention to complete the job in a reasonable amount of time. The point is that

needing a program that works fast is no excuse for ignoring the other aspects of good programming. A program cannot be considered as a good program unless it is written in a way that makes it easy to understand. Some of the reasons for this emphasis on programs that are easy to understand have been explained in earlier chapters but, generally speaking, a program that is easy to understand is less likely to contain hidden errors and will be easier to update and modify in the future.

The most important components of producing programs which are easy to understand are a simple flow of control and the use of subroutines to encapsulate the different actions of a program. The subject of how to make the flow of control easy and clear to understand is still a slightly controversial subject with some factions claiming that the arch-enemy of a clear flow of control is the GOTO statement. This is a matter of opinion and it is much better for you to appreciate the need for a simple flow of control than be told never to use the GOTO – advice that you are sure to be given at some time! In some cases, the GOTO statement is the clearest and most unambiguous way of altering the flow of control.

When it comes to the use of subroutines, a good program should be understandable at a number of levels. When you first look at it you should be able to see its overall structure from the way that its main subroutines are called. You should then be able to find out about its more detailed workings by examining the lines of BASIC that make up the main subroutines and then by examining the subroutines that are used within the main subroutines and so on. A good program should have a structure like the layers of an onion!

If a program is well written in the way described above then its workings should be almost self-explanatory but even in the clearest of programs there is often the need to supply some extra information about how things work. For this reason BASIC includes the REM statement. The BASIC command REM is the simplest of all in that it does absolutely nothing! Its only purpose is to allow you to include comments that are not part of a program. For example, you should include in every program that you write a first line that tells you what the program is intended to do. In the following program:

```
10 REM This is a test
```

the REM alerts the Oric to the fact that what follows isn't to be taken as a line of BASIC but as a note to any humans who might read the program. REM statements can be included at any point in a program and can serve as a note to the programmer of what the



program is doing at its various stages. This is a useful facility, especially if you intend other people to scrutinise your programs or want to re-use them yourself after an interval. You might think it strange that such a simple command has been left to the last chapter of a book on BASIC. The reason for this is that, although REM is a simple command, it's not until you reach a certain level of understanding that you can see what better programming is all about. The correct use of the REM statement is a sure sign that you are no novice programmer!

## Error trapping

Writing good programs, in the sense of good style, is only one side of the coin – the programmer's side. Program structure is an internal consideration that should interest other programmers but, apart from reducing the number of bugs, is of no interest to users. Users only see the *outside* of your programs and this is yet another aspect of writing good programs. A program that is easy to use and difficult to *crash*, that is to stop running by typing in incorrect data, is said to be *user-friendly*. It is difficult to explain in general terms how to write user-friendly programs because it depends very much on the individual program and who the intended users are. However, there are a number of simple things that you can do to make sure that your program is relatively uncrashable. In particular you should check that any input data is in the range you would expect. For example, rather than:

```
10 INPUT "ANSWER THE QUESTION YES OR NO";A$
20 IF A$="YES" THEN GOSUB yyyy
30 GOSUB nnnn
```

where subroutine yyyy is the subroutine that handles the 'yes' answer and subroutine nnnn handles the 'no' answer, use:

```
10 INPUT "ANSWER THE QUESTION YES OR NO";A$
20 IF A$="YES" THEN GOSUB yyyy:GOTO 60
30 IF A$="NO" THEN GOSUB nnnn:GOTO 60
40 PRINT "PLEASE ANSWER YES OR NO"
50 GOTO 10
60 rest of program
```

In other words, check that the answer to a question is one of the allowed answers rather than just assuming that if it isn't one answer

it must be the other. Checking that a numeric input is in the correct range is even easier. For example if the input should be in the range 1 to 10 then use:

```
10 INPUT "ENTER A VALUE IN THE RANGE 1 TO 10";A
20 IF A<1 OR A>10 THEN PRINT "A NUMBER
   BETWEEN 1 AND 10 PLEASE":GOTO 10
30 rest of program
```

Whenever you get data from the outside world into a program – check that it is reasonable.

However even if you check all the input data values and make sure that they are in the correct range you can still get errors that cause the program to crash. For example, during the course of a perfectly legal calculation you might try to divide by zero or the result might become too big. Such errors are known as *run time errors* because they are difficult to spot just by looking at the way that the program is written. Rather than just allowing such run time errors to cause the program to crash, it is possible to detect or *trap* most of them before they occur. For example if there is any chance that a sum which involves division could fail because of attempting to divide by zero then simply work out the sum in three stages. That is instead of:

$$A=(B*3-C)/(C*4-8)$$

where in practice the two expressions in brackets could be even more complicated use:

```
D=(C*4-8)
IF D=0 THEN PRINT "CANNOT WORK IT OUT
FOR THESE VALUES":GOTO xxx
A=(B*3-C)/D
```

Another run time error that occurs quite often is produced by trying to move the graphics cursor outside the screen area. Trapping this run time error is particularly simple for the CURSET command. Each CURSET command should be preceded by:

```
IF X<0 THEN X=0
IF X>239 THEN X=239
IF Y<0 THEN Y=0
IF Y>199 THEN Y=199
```

Of course in practice this just isn't possible because your program would almost certainly be too big and too slow as a result! It is much more difficult to check if commands such as CURMOV and DRAW

are going to take the cursor off the screen because of their use of relative co-ordinates but it is possible. It just isn't practical to try to check for all run time errors and so, in Oric BASIC at least, the possibility of crashing a program has to be admitted. In general it is very difficult to make a program completely crash proof, but if you design the error handling part of your program as you go along rather than adding it when the program is almost finished you will find it easier. Finally, it is important to realise that error handling is one of the most difficult areas of programming and it is certainly to be left until you are happy with other aspects of BASIC and writing programs.

### **Where next?**

The real route to learning how to program is to write programs! This said it does help to know something of the theory that lies behind programming and how other programmers go about the task. A good way to learn is to read other people's programs and try to find out how you could have done better. Plenty of examples of graphics games programs that you might enjoy playing and improving on are contained in *The Oric Book of Games*, by Mike James, S. M. Gee and Kay Ewbank, published by Granada. Once you have tackled a few larger programs of your own then it is time to look at the more advanced aspects of BASIC programming which you will find further explained in *The Complete Programmer*, by Mike James, also published by Granada. Whatever you do it is important not to lose the sense of adventure and experimentation that makes programming challenging and enjoyable!

## Appendix

# Bugs in the Oric's ROM and how to deal with them

There are a number of known bugs in the Oric's BASIC ROM. Most of them are not serious and it is not difficult to find ways around them. Many of the problems have been mentioned in the text along with ways of fixing them. It is, however, worth gathering them all together in one place and this also provides an opportunity to list one or two of the more obscure bugs. Wherever possible you should attempt to write programs that get round the bugs in such a way that the program will still run even if the Oric's ROM is replaced and the bugs permanently fixed.

### **TAB**

The most obvious bug is the fact that the TAB function doesn't work properly. To cure the bug you can use TAB(X+12) in place of TAB(X) but this will result in a program that doesn't work on a machine that has a corrected ROM. The best fix is to avoid the use of the TAB function altogether and use SPC(X), which in Oric BASIC has exactly the same action as TAB(X).

### **STR\$**

The STR\$ function adds a control character to the front of the string that it returns as its result. However, it only misbehaves if the number X in STR\$(X) is positive. The most obvious fix is to remove the first character from the front of the string using A\$=RIGHT\$(A\$,LEN(A\$)-1) but this doesn't work if the number in the STR\$ function is negative and gives the wrong result if used with a ROM that doesn't have the bug. The best solution is to test for control codes at the front of the string and only remove them if they are present! That is, use:

```
A$=STR$(X)
IF ASC(A$)<32 THEN A$=RIGHT$(A$,LEN(A$)-1)
```

## **PRINTing ESC in column one**

Normally printing the ESC character, CHR\$(27), doesn't store the ESC character at the current printing position but causes the Oric to interpret the next character as an attribute code which is stored at the current printing position. On many Orics this only works once the printing position has gone beyond column one. If you PRINT CHR\$(27) and the current printing position is column one then attribute code 27 is incorrectly stored at column one with results that were never intended! The fix is quite simple and works for all Orics – don't print attribute codes using the ESC character to column one, use the PLOT command instead!

## **FILL command**

The FILL command should move the graphics cursor to the last dot in the last character cell that it affects. However, in practice it doesn't update the graphics cursor at all. The fix is never to assume the position of the graphics cursor following a FILL command but use CURSET to position it.

## **POKE**

The POKE command will not accept hex values after the comma. The fix is to simply convert all hex values to decimal by storing them in a variable first. That is, rather than POKE 223,\33 use:

```
TEMP=#33
POKE 223,TEMP
```

## **Spurious printer characters**

The printer port randomly outputs squiggles, that is CHR\$(7F) characters. The fix is to turn off interrupts using:

```
CALL Ø6CA
```

before printing using LLIST and then restore interrupts using:

```
CALL 0804
```

afterwards.

### **String or array corruption**

This is caused by incorrect setting of HIMEM on power up. The fix is explicitly to set HIMEM using:

```
HIMEM \97FF
```

### **Program loading**

The Oric will sometimes load a program successfully but with the end of the program replaced by lines of U's. This is caused by a line linkage error which the Oric's tape handling doesn't seem to detect. Sometimes entering a dummy line such as 1 REM at the start of the program and then deleting it will force the BASIC ROM to relink the program. However, sometimes the problem is so bad that this doesn't work. The fix under these circumstances is extremely complicated and beyond the scope of this appendix. The best course is to avoid relying on a single copy of your program by saving at least twice, including once at the lower baud rate.

# Index

ABS, 66  
AND, 140  
arithmetic expression, 17  
arithmetic function, 66  
array, 57  
array element, 57  
arrow keys, 72  
ASC, 69  
ASCII code, 69  
assignment statement, 15, 50  
ATN, 68

binary, 143  
Boolean logic, 140  
brackets, 18

CALL, 145, 157  
cassette recorder, 4  
CHAR, 132  
chromatic scale, 109  
CHR\$, 69  
CIRCLE, 127  
CLEAR, 146  
colon, 44  
condition, 31  
conditional expression, 31  
constants, 19  
CONT, 9  
COS, 68  
CPU, 2  
crash proof programs, 153  
CTRL, 11, 29  
CURMOV, 124  
CURSET, 122  
cursor keys, 11

DATA, 61  
debugging, 149  
DEEK, 145  
DEF FN, 74

default flow of control, 28  
deferred mode, 8  
DELeTe key, 8, 11  
DIM, 57  
DOKE, 145  
double height characters, 99  
DRAW, 124  
dummy parameter, 74

END, 47  
enumeration loop, 40  
ENVELOPE, 115  
error trapping, 154  
ESC, 11, 89, 157  
exclusive or, 142  
EXP, 66  
EXPLODE, 112  
exponential number, 66

false, 33, 141  
FALSE, 143  
FILL, 128, 157  
flag, 61  
flashing characters, 100  
flow of control, 28  
FN, 74  
FOR ... TO, 40  
fractions, 56  
FRE, 73  
function, 64

garbage collection, 73  
GET, 71  
GOSUB, 76, 146  
GOTO, 29, 76, 152  
GRAB 133  
graphics characters, 93  
graphics cursor 123

hexadecimal, 144

- HEX\$, 69
- HIMEM, 146
- HIRES, 121
- IF ... THEN, 31, 141
- IF ... THEN ... ELSE, 46, 77, 141
- immediate mode, 8
- inclusive or, 142
- index variable, 41
- infinite loop, 29
- INK, 92, 128
- inner loop, 44
- INPUT, 20, 50, 72
- input device, 2
- input prompt, 22
- INT, 67
- integer, 56, 67
- inverse colours, 101, 132
- iteration, 30
- keyboard, 6
- KEY\$, 71
- LEFT\$, 54, 69
- LEN, 54, 69
- LET, 15
- line number, 8
- LIST, 8
- literal string, 22
- LLIST, 158
- LN, 67
- LOG, 67
- logarithm, 67
- loop, 29
- LORES, 93
- machine code, 145
- MID\$, 52, 70
- MUSIC, 105
- NEW, 8
- NEXT, 41
- NOISE, 113
- NOT, 141
- null string, 53
- numeric variable, 14
- OR, 142
- order of evaluation, 18
- outer loop, 44
- output device, 2
- PAPER, 92, 128
- parameter, 65
- PATTERN, 128
- PEEK, 144
- PING, 112
- PLAY, 105
- PLOT, 82, 86, 89, 102
- POINT, 133
- POKE, 144, 146
- POP, 146
- PRINT, 15, 50, 80, 89
- pseudo random number generator, 71
- PULL, 146
- radians, 68
- raise to a power, 17
- RAM, 3
- random numbers, 71
- randomness, 71
- READ, 61
- real numbers, 56
- relation, 32, 54
- RELEASE, 133
- REM, 152
- REPEAT ... UNTIL, 39, 147
- rest, 112
- RESTORE, 62
- RETURN, 8, 11, 76, 146
- RIGHT\$, 54, 70
- RND, 70
- ROM, 4, 156
- RUN, 8, 11
- run time errors, 153
- SCRN, 119
- select, 34
- semicolon, 23
- serial attributes, 5, 79, 89
- SGN, 67
- SHOOT, 112
- simple variable, 14
- SIN, 68
- skip, 34
- SOUND, 107
- SPC, 81
- SQR, 65, 67
- STEP, 42
- STOP, 47, 150
- string, 22, 50
- string concatenation, 52
- string constant, 50
- string expression, 51
- string function, 69
- string variable, 50
- STR\$, 70, 84, 156



subroutine, 75  
substring, 52  
substring extraction, 52  
substring replacement, 52  
substring searching, 54

TAB, 81, 156  
TAN, 68  
TEXT, 122  
text cursor, 86  
trigonometrical functions, 68  
TROFF, 150  
TRON, 150  
true, 33, 141  
TRUE, 143  
truth table, 142

tunes, 109  
  
unary minus, 17  
until loop, 39  
user-defined characters, 97  
user-defined functions, 74  
user-friendly programs, 153  
USR, 145

VAL, 70  
variable, 13  
variable names, 14, 50

WAIT, 106  
while loop, 39

ZAP 112











**THE ORIC-1  
PROGRAMMER**

M. James and S. M. Gee  
0 246 12157 2

**ORIC MACHINE CODE  
HANDBOOK**

Paul Kaufman  
0 246 12150 5

**The TI 99/4A****GET MORE FROM  
THE TI99/4A**

Garry Marshall  
0 246 12281 1

**The VIC 20****GET MORE  
FROM THE VIC 20**

Owen Bishop  
0 246 12148 3

**THE VIC 20  
GAMES BOOK**

Owen Bishop  
0 246 12187 4

**The ZX Spectrum****THE ZX SPECTRUM  
And How To Get  
The Most From It**

Ian Sinclair  
0 246 12018 5

**THE SPECTRUM  
PROGRAMMER**

S. M. Gee  
0 246 12025 8

**THE SPECTRUM  
BOOK OF GAMES**

M. James, S. M. Gee  
and K. Ewbank  
0 246 12047 9

**INTRODUCING  
SPECTRUM  
MACHINE CODE**

Ian Sinclair  
0 246 12082 7

**SPECTRUM GRAPHICS  
AND SOUND**

Steve Money  
0 246 12192 0

**THE ZX SPECTRUM  
How to Use and  
Program**

Ian Sinclair  
0 586 06104 5

**Learning is Fun!  
40 EDUCATIONAL GAMES  
FOR THE SPECTRUM**

Vince Apps  
0 246 12233 1

**The ZX81****THE ZX81  
How to Use and  
Program**

S. M. Gee and  
Mike James  
0 586 06105 3

**Which Computer?****CHOOSING A  
MICROCOMPUTER**

Francis Samish  
0 246 12029 0

**Languages****COMPUTER  
LANGUAGES AND  
THEIR USES**

Garry Marshall  
0 246 12022 3

**EXPLORING  
FORTH**

Owen Bishop  
0 246 12188 2

**Machine Code****Z-80 MACHINE  
CODE FOR HUMANS**

Alan Tootill and  
David Barrow  
0 246 12031 2

**6502 MACHINE  
CODE FOR HUMANS**

Alan Tootill and  
David Barrow  
0 246 12076 2

**Using Your Micro****COMPUTING FOR  
THE HOBBYIST AND  
SMALL BUSINESS**

A. P. Stephenson  
0 246 12023 1

**DATABASES FOR  
FUN AND PROFIT**

Nigel Freestone  
0 246 12032 0

**SIMPLE INTERFACING  
PROJECTS**

Owen Bishop  
0 246 12026 6

**INSIDE YOUR  
COMPUTER**

Ian Sinclair  
0 246 12235 8

**Programming****THE COMPLETE  
PROGRAMMER**

Mike James  
0 246 12015 0

**PROGRAMMING  
WITH GRAPHICS**

Garry Marshall  
0 246 12021 5

**Word Processing****CHOOSING A  
WORD PROCESSOR**

Francis Samish  
0 246 12347 8

**WORD PROCESSING  
FOR BEGINNERS**

Susan Curran  
0 246 12353 2

## HOW TO BECOME AN ORIC VIRTUOSO!

Now the Oric is yours you will want to complete your programming apprenticeship as quickly as possible. This book is aimed at every beginner, young or old, at home or in business. It teaches BASIC from first steps to a high level of competence, paying special attention to the Oric's graphics and sound capabilities. Most chapters contain at least one complete program listing to enjoy as you learn, and many examples of shorter programs are provided to help you master the techniques. Everything you need to know is set out clearly and logically so that you are soon in complete control of this powerful machine.

### *The Authors*

Mike James is the author of *The BBC Micro: An Expert Guide*, and both he and S M Gee have written several other very successful books on programming. They are both regular contributors to the monthly computing magazines.

Also from Granada

### **THE ORIC-1**

#### **And How To Get The Most From It**

*Ian Sinclair*

0 246 12130 0

### **THE ORIC BOOK OF GAMES**

*Mike James, S M Gee and Kay Ewbank*

0 246 12155 6



GEE AND JAMES

# THE ORIC PROGRAME

GRANADA